Cohesion in Computer Text Generation: Lexical Substitution

by

Robert Alan Granville

May, 1983

This research was supported (in part) by the Office of Naval Research contract N0014-80-C-0505 and (in part) by the National Institutes of Health Grant No. 1 P01 LM 03374-04 from the National Library of Medicine.

Laboratory for Computer Science

MASSACHUSETTS INSTITUTE OF TECHNOLOGY
Cambridge, Massachusetts 02139

Cohesion in Computer Text Generation: Lexical Substitution

by

Robert Alan Granville

Submitted to the Department of
Electrical Engineering and Computer Science
in Partial Fulfillment of the
Requirements of Master of Science

Abstract

This report describes *Paul*, a computer text generation system designed to create cohesive text. The device used to achieve this cohesion is lexical substitution. Through the use of syntactic and semantic information, the system is able to determine which type of lexical substitution will provide the necessary information to generate an understandable reference, while not providing so much information that the reference is confusing or unnatural.

Specifically, *Paul* is designed to deterministically choose between pronominalization, superordinate substitution, and definite noun phrase reiteration. The system identifies a strength of antecedence recovery for each of the lexical substitutions, and matches them against the strength of potential antecedence of each element in the text to select the proper substitutions for these elements. There are five classes of potential antecedence, based on the element's current and previous syntactic roles, semantic case roles, and the current focus of the discourse. Through the use of these lexical substitutions, *Paul* is able to generate a cohesive text which exhibits the binding of sentences through presupposition dependencies, the marking of old information from new, and the avoiding of unnecessary and tedious repetitions.

Thesis Supervisors:

Peter Szolovits
Associate Professor of Computer Science

Robert Berwick
Assistant Professor of Computer Science

Keywords: Natural Language, Natural Language Generation, Utterance Realization, Cohesion, Endophoric Reference, Lexical Substitution, Synonyms, Superordinates, General Nouns, Pronominalization, Focus, Distance

Acknowledgments

I would like to thank everyone who contributed to this work, and everyone who aided, abetted, encouraged, cajoled, or in any way assisted me during this project:

I would like to personally thank Pete Szolovits and Bob Berwick for their exceptional advice and encouragement, their invaluable contributions in clearing my thinking processes and improving my writing, and for reading drafts with incredible speed when time was of the essence.

I owe a huge debt to the Clinical Decision Making Group for providing insights, encouraging me when I needed it, putting me on the right track every time I strayed, and adding a little humor to the work day.

I would like to remember all my past co-workers over the past few years who have helped me in thinking about this work and the direction it should take. I especially thank Lance Miller, Amy Zwarico, Michal Blumenstyk, Mugsy, Lightfingers, Machinegun, Maddog, and the whole Heidorn gang.

I feel I owe a special personal debt to George Heidorn for his encouragement, his sacrifice of personal time for my part, his keen insights and suggestions, and for his tolerance of the Kid's capricious whims.

Without personal friends to give me support and perspective, this thesis and I would not have survived each other:

I would like to thank the Ashdown Irregulars for their sparkling conversational wit, their charming dinner companionship, and their always just being there when I needed them. I especially would like to thank Brian Oki, Richard Sproat, and Monty McGovern for tolerating me when I was silly, and supporting me when I wasn't.

I would like to thank the MIT Community Players for giving me an outlet and constantly reminding me that the problem set will be there tomorrow, but this flat has to be built tonight. I owe a special debt to Amy Schrom, Ronni Marshak, and especially Robin Nelson for providing me with wonderful shoulders to cry on more times than I can remember.

I owe much to Corine Bickley for telling me that everything is going to be all right, and for keeping my attitude properly adjusted.

My dictionary defines *faith* as "belief without proof, confidence, reliance, loyalty:"

To Karen Jensen, who had faith in me long after I had ceased to, and who wouldn't let me give up on myself. I cannot express the depth of my debt toward, and the magnitude of my appreciation and affection for, Karen. If there is anything good in this thesis, it is directly attributable to her, while all shortcomings are due to the shortcomings of the author. Karen has been the best of teachers, advisors, coworkers, confidants, companions, and most importantly, friends.

To my parents, who believe in me and my work without understanding, which is the most profound act of faith. This, and everything I do, is humbly dedicated to them.

Table of Contents

Table of Figures

1. Introduction

1.1. Statement of the Problem

The need for computer systems to generate acceptable text in a natural language such as English is constantly increasing. While computer text generation is an interesting problem in itself, other types of systems have found a requirement for the ability to communicate in a natural language. This is especially true for computer systems that attempt to address human factor issues, that is, systems that strive to make computers easier to use, or more "friendly," especially for people outside the field of computer science. One obvious way to enhance this "friendliness" is to have the computer communicate in a language that is easy for the user to understand, her own natural language, rather than in a language that is easy for the machine to understand, a programming language, that requires an effort on the user's part to learn. If we hope to build systems that gain general acceptance and widespread usage, we must be willing to incorporate natural language communication into these systems.

This report describes *Paul*, a computer text generation system designed to create cohesive text. The device used to achieve this cohesion is lexical substitution. Through the use of syntactic and semantic information, the system is able to determine which type of lexical substitution will provide the necessary information to generate an understandable reference, while not providing so much information that the reference is confusing or unnatural.

Specifically, *Paul* is designed to deterministically choose between pronominalization, superordinate substitution, and definite noun phrase reiteration. The system identifies a strength of antecedence recovery for each of the lexical substitutions, and matches them against the strength of potential antecedence of each element in the text to select the proper substitutions for these elements. There are five classes of potential antecedence, based on the element's current and previous syntactic roles, semantic case roles, and the current focus of the discourse. Through the use of these lexical substitutions, *Paul* is able to generate a cohesive text which exhibits the binding of sentences through presupposition dependencies, the marking of old information from new, and the avoidance of unnecessary and tedious repetitions.

In natural language generation, there are at least six criteria that computer output must meet before it could be considered acceptable.

1. The generated text cannot be canned.

2. The generated text must be based on an arguably correct semantic representation.

3. The generated text must exhibit cohesion.

4. The generated text must be comprehensible.

5. The generated text must not have erroneous connotations.

6. The generated text must not violate the intended style and mood.

The use of previously prepared strings of text, known as *canned text*, is possibly the simplest and most obvious way to have the computer respond in natural language. Having the computer merely display the appropriate stored text is adequate in many applications, such as in error messages. However, in addition to being a relatively uninteresting approach, the use of canned messages has severe limitations [30]. All possibly desired messages must be anticipated in advance, which is not always (indeed not usually) a feasible feat. While the use of slot filling, strategically placing variables in the text (such as the name and address of the recipient of a form letter), allows an additional freedom, the general outline as well as the bulk of the text is still dictated in advance, and therefore fixed. Each such invariant text must be composed by humans in advance and permanently stored on the system. The occurrence of a new situation, or even just a new and unexpected variation on an anticipated situation, would require the creation of an entirely new text. Furthermore, the computer cannot aid in any meaningful way in the creation of these texts, and the complete lack of generality in this method prevents an implementor from gaining any benefits from economy of scale. Having written and entered a score of error messages will not ease or affect the labor of writing and entering an additional score.

The current alternative to canned text is to translate knowledge structures into a natural language. The temptation here is to follow the expedient of developing knowledge structures and translation rules that are adequate for a chosen domain. While proper use of constraining one's domain is beneficial, and with the current state of the art probably necessary, taking undue advantage of the constraints and avoiding generality for the sake of convenience will produce systems that apply *only* to their specific domains, and do nothing to further knowledge of natural language systems in general. Rather than being merely adequate, systems should be based on knowledge structures and translating rules that are general enough to adequately produce text in several domains. In order to be able to achieve this goal, systems must have knowledge structures and translating rules that are linguistically justified [30]. The generated text must be based on an arguably correct semantic representation.

In order for a string of sentences to be considered a text in natural language, those sentences must exhibit cohesion, an interdependence between the sentences created by causing the interpretation of some elements to be dependent on other elements [11].

Sam is upset. <u>He</u> can't go to Gertrude's party.

The interpretation of *he* in the second sentence is dependent on *Sam* in the first.

Text without cohesion has the stilted and awkward feel of an elementary school primer, and certainly doesn't sound intelligent. Such text would be unacceptable for most systems today that require a natural language capability. The generated text must exhibit cohesion.

While cohesion is necessary for acceptable text generation, it is not sufficient. It is possible through the injudicious use of certain cohesive devices (such as pronouns) to render a text completely unintelligible. If elements appear whose interpretations depend on other elements that *don't* appear, either because the program mistakenly neglected to put these elements in, or worse yet, replaced them with additional cohesive devices, the resulting text will be ludicrous and serve no useful purpose. (The classic example of carrying this problem to an extreme is the verse "evidence" read by the White Rabbit in Chapter XII of Lewis Carroll's *Alice's Adventures in Wonderland*[1].) A computer generated text that no one can understand is simply not a text. The generated text must be comprehensible.

Additionally, if the program has an option in word selection from its vocabulary, care must be taken in this selection process. In addition to meaning, most words in any natural language have connotations and implications associated with them. For instance, consider the synonyms one would find in a standard thesaurus for the word "smell." The synonyms listed could be of a favorable nature, such as "emanation," "fragrance," or "aroma"; they could be of a neutral sense, as in "odor," "smell," or "scent"; or they could express distaste, as with "stench," "stink," or "foulness." All these words should be achievable from the structure representing the concept SMELL, but clearly they are not interchangeable. The program must have some means of selecting words with the desired implications, or at least avoiding words that have blatantly wrong implications. The generated text must not have erroneous connotations.

In the same vein as connotation, words have senses of style and mood that must be considered. Certain vernacular phrases, terms of endearment, and other ways of expressing familiarity would be inappropriate in a medical diagnosis or a formal business letter, but they would not only be acceptable but expected in a close personal communication. A letter to a good, steady customer who has apparently forgotten a small bill should not have the same tone as a letter to a person who is considerably behind in her accounts and has ignored several communications to that effect. Vocabulary selection and grammatical constructions can have a large impact on the mood and style the text is going to have, and this impact needs to be taken into account. The generated text must not violate the intended style and mood.

[1]The text of this evidence appears as an appendix.

1.2. Cohesion

A set of sentences must exhibit cohesion to be considered a well-formed text. There are several necessary functions that are provided by this cohesion. The first one, already mentioned briefly in the previous section, is that without cohesion a text is awkward and appears unintelligent. An example might help demonstrate this.

```
T1.1: John went to the store.
      John bought a kite.
      John went home.
```

The sentences of T1.1 appear to be isolated. As speakers of English, we want relationships between the sentences, but there is nothing in T1.1 to make these relationships clear. The average reader would be unhappy calling T1.1 well-formed text.

However, this is not the only effect cohesion, or its lack, has on text. The speaker[2] wants her thoughts understood, and the listener wants to understand the thoughts being conveyed. Cohesive devices can help make this task easier by distinguishing old information from new [15, 16]. Consider the following example, which is T1.1 with a simple modification.

```
T1.2 A boy went to the store.
     A boy bought a kite.
     A boy went home.
```

T1.2 is even more objectionable to the average reader than was T1.1. Not only are the intersentential relationships not explicit, but the text is ambiguous. Are we referring to one boy who performed all the actions in the text, or three separate boys, one who went to the store, one who bought a kite, and one who went home? T1.2 demonstrates that cohesion is more than a device to make text more elegant or pleasing. It is necessary for marking old information from new, for distinguishing references to items and ideas that have been already mentioned from those that are being introduced for the first time. Since the speaker's goal in most discourse is to either elaborate the details of a specific idea or item, or to explain the relationship between an item or idea known by the speaker to one that is new to the speaker [15, 17, 16], this ability to distinguish new from old is essential. Obviously, a computer system that generates text must also be able to perform cohesion in order to generate understandable text.

[2]Or *writer.* For the larger part, we will not distinguish between written and oral text in this section.

1.3. Lexical Substitution

The cohesive devices that will be discussed in this report are collectively known as *lexical substitution* [11]. Lexical substitution includes *general noun substitution*, the replacement of a specific reference to an entity with one of the so-called *general nouns* [11] (such as *man, woman, boy, girl* for humans, *creature, beast* for animals, *matter, affair* for inanimate abstracts), *synonymous substitution*, the use of synonyms, and *superordinate substitution*, the replacement of specific references with words with a more general meaning (i.e. "vehicle" as a replacement for "car"). *Pronominalization*, the use of pronouns, is treated as a special form of lexical substitution, and is included in this report. Finally, *definite noun phrases*, using the definite article "the" as opposed to the indefinite article "a," can be used as a last resort when the four lexical substitutions listed above cannot be applied.

Particularly in superordinate substitution, the danger exists that the word selected could have more than one referent in the text. For example, if writing about a Volvo and a Ford, a reference to "the car" is ambiguous. One way to handle this problem is by disambiguating the superordinate selection as much as is necessary, but no more than is necessary [9]. If we were writing about a green Volvo and a red Ford, while "the car" would be unacceptably ambiguous, "the green car" or "the foreign-made car" would be clear. However, "the green foreign-made car" would be blocked because it gives more information than is necessary to disambiguate the reference.

Since the vocabulary hierarchy is semantically based, synonym substitution is fairly straightforward. Care must be shown, though, in order that erroneous connotations are not created, nor style and mood violated. Rather than simply having a list of words that express a concept and selecting from that list, the words must be partitioned into distinct (and possibly disjoint) sets based on their connotations. Furthermore, each distinct set must be further partitioned by the style and mood effects the individual words exhibit. Such a partitioning yields lists of words that are truly synonymous and can be readily substituted in the text without incorrectly impacting style, mood, or connotative considerations.

1.4. The Approach of This Report: *Paul*

Paul is a natural language generation program initially developed at IBM's Thomas J. Watson Research Center this past summer as part of the ongoing Epistle project [14, 20]. One of the ultimate goals of the Epistle project is to generate business letters from a one or two sentence description of the topic, and access to a knowledge base containing information about the recipient, the nature of the business, and related business correspondence [14]. *Paul* was designed as a first step to generate text from the appropriate knowledge structures once these structures have been created. The system, written in NLP [13], accepts knowledge structures in the form of NLP records and translates them, through NLP rules, into multisentential text. Such a natural language generation system following the six criteria explained above has been achieved by expanding and refining the *Paul* system.

All NLP programs manipulate, alter, and create NLP records as the basic primitive data structure. These records are very similar to frames [36].

The NLP rules that make up the program that translates the records into English text are based on *augmented phrase structure grammar* [13, 45]. Augmented phrase structure rules are very similar to the concept of phrase structure rules [4] that linguists are familiar with. The chief difference is that specifications can be placed on the structures being manipulated. Since these specifications are created by the user and can contain any information desired, the rules need not be strictly syntactic, but can reflect semantic and pragmatic information as well. A subset of NLP, that which is necessary for natural language generation, has been implemented at MIT in MACLISP for *Paul.*

The emphasis of the *Paul* system is in research of *discourse phenomena,* the study of cohesion and its effects on multisentential texts [11, 38]. Text generation can be divided into two distinct subtasks [30, 27, 35]. The first subtask is to create the knowledge structures that will be used, ensuring that these structures are correctly ordered and contain the desired knowledge. In other words, this subtask is to determine *what* the text is to say. This subtask will be called *utterance*[3] *planning* in this paper. The second subtask is to take the created knowledge structures and translate them into the target natural language, taking care that the six criteria discussed above, especially those concerning style and cohesion and their effects, are met. In other words, this subtask is to determine *how* to say it. This subtask will be called *utterance realization* in this paper. *Paul* is an utterance realization system.

By the very nature of the fact that *Paul* translates knowledge structures into English, the system does not make use of canned text in any form. Therefore, the first criterion that the generated text not be canned is met by *Paul.*

For a natural language generation system to be based on an arguably correct representation, its knowledge structures must be linguistically motivated. In *Paul,* knowledge is represented in NLP records through the use of a case frame [6] formalism, where each case corresponds to an NLP record attribute. Furthermore, records are used to set up a semantic hierarchy of vocabulary. Words are currently arranged in a *superset* hierarchy, similar to AKO links [46]. The refinement of *Paul* has a fairly extensive overhaul of this

[3]By *utterance* we again mean both spoken and written natural language production, rather than that restricted to oral.

hierarchy. Rather than having words as the main entries in the vocabulary data base, *conceptual* or *primitive* structures [39, 25] contain the semantic information necessary for initial selection of vocabulary. Separate word entries contain morphological information, such as irregular plural or past tense formations. By keeping conceptual information separate and distinct from morphological knowledge, two major advantages are gained. First, the program is free to make vocabulary selections to express the desired concepts, rather than have these selections made explicitly for the program. Second, by having morphological information separate, generalities can be captured that otherwise might be missed. As an example, consider the word "have." "Have" has at least three very distinct meanings: as an auxiliary verb ("I *have* finished."), as a verb meaning to possess ("I *have* it."), and as a verb meaning to cause someone to do something ("I *have* a maid come in twice a week."). Each distinct meaning of "have" should have its own entry in the semantic hierarchy. However, the word "have" has only one conjugation, regardless of its current semantic use. This irregular conjugation must be made explicitly known to an English language generation system. If a morphological entry for "have" did not exist distinct from the semantic entries, the information would have to be repeated for each semantic entry. Having a separate entry captures the necessary morphological generality.

Deciding when lexical substitution would be proper, and which of the several devices should be used is a difficult task, although controlling such a choice is a very important requirement for the use of any cohesive device. Abusing the text by overusing cohesive devices will yield output suitable only for humorous purposes. Intelligibility must be preserved. Furthermore, consideration must be given to the connotations behind any words selected to create cohesion, as well as their effects on the desired style and mood of the text.

The problem in using these cohesive devices is that it is necessary to guarantee that they are understandable. That is, since these items refer *anaphorically* [38, 11] to a previously mentioned item, called the anaphor's *antecedent* [7, 24], it is required that the anaphor can be unambiguously related back to its antecedent. Otherwise, unintelligible text may be generated.

Investigation into anaphora resolution has been performed in the pursuit of natural language understanding [5, 10, 41]. Many of these works propose using *focus* or *theme* [11] as a basis to restrict or predict the eligible candidates for the antecedent of a given anaphor ([41] particularly). Informally, focus is what a sentence is about, that is, the central point of the utterance. In [41], each noun phrase in a sentence is ranked for its potential as the focus. Then, when an anaphor occurs, the ranked list is tested in order for syntactic, semantic, and pragmatic acceptance. The first item in the ranked list that passes these criteria is assumed to be the antecedent for the anaphor, and is confirmed as the focus.

Focus was also used rather successfully in generation, notably by McKeown [35]. Her TEXT system, designed to address problems in utterance planning, uses focus to restrict the system's options in what should be said next. Focus is used to eliminate choices that have no bearing on the current focus of the discourse, and furthermore, focus is used and shifted to determine which of the various relevant items will actually be generated.

Unfortunately, a theory of anaphora generation involving only focus is inadequate. While a sentence has only one focus, *every* entity referred to within a sentence must be somehow marked as old information in later sentences, *not* just the focus of the sentence. Consider the following example:

```
T1.3 1.  John sent a letter to Mary.
     2a. He wanted to see her.
     2b. She was glad to receive it.
     2c. He wrote it by hand.
```

The focus of the sentence in T1.3-1 is *a letter*, as it is the object receiving the action of being written. A lexical substitution theory which allows replacement of foci only would allow only the noun phrase *a letter* to be pronominalized in a following sentence. But T1.3-2a, T1.3-2b, and T1.3-2c are all acceptable sentences to follow T1.3-1, even though each has a pronominalization of noun phrases other than the focus of T1.3-1. In fact, T1.3 demonstrates *each* possible pair of noun phrases pronominalized. Clearly, a theory for lexical substitution based on such a narrow view of focus is inadequate.

Paul controls lexical substitution through the use of *minimal features*. Each noun phrase that is a candidate is identified, and the minimal amount of information that is required to make an understandable reference is calculated. *Paul* then determines which of the various forms of lexical substitution (including no lexical substitution) provides the minimal features to keep the text clear.

Rather than isolating one entity in a sentence and labeling that as the focus eligible for lexical substitution, *all* entities mentioned in the sentence are labeled as *focal points* of the sentence and therefore subject to lexical substitution. The distinction here is that the data base from which the semantic representation is created has a good deal more information than is being expressed in the sentence. For instance, for sentence T1.3-1, the data base could conceivably have knowledge about the size of John, his age, the color of his hair, etc., and of course, the same kinds of information would be stored in the entry for Mary. However, most of these items were screened out during the utterance planning phase of generation. These items are *not* eligible for lexical substitution, and references to them in future sentences must be explicit. The points from previous sentences *are* eligible for lexical substitution.

The various forms of lexical substitution, however, are not interchangeable, because they offer differing levels of difficulty in *antecedence recovery*. Pronouns are the most difficult to recover, because they convey the least amount of information. The only knowledge explicitly given by a pronoun is number and gender (*if* singular). General nouns offer little more except for the general class the antecedent belongs to.

Superordinate substitution is fairly explicit, especially with the proper choice of descriptive adjectives to disambiguate the reference. Synonyms are the strongest reference, since they are not true examples of anaphor, but merely a device to avoid unnecessary and tedious repetition. And of course, since definite noun phrases are not a form of substitution at all, there is no problem of antecedence recovery.

We can control the selection of lexical substitution devices by determining the minimal features required to provide an understandable reference, and which lexical substitution will provide these minimal features. This is done by ranking the focal points of a sentence by their *strength of potential antecedence*. This ranking is based on several factors, including both syntactic and semantic information. These factors are the point's position in an *expected focus list*, the number and gender of the item as well as the numbers and genders of all previously mentioned items, the distance between the current mentioning of the item and the last previous reference, the syntactic role the item played in the last reference as well as its current syntactic role, and whether an item is a part of a previously mentioned item or a member of a previously mentioned set. These factors allow us to identify the various *classes* of strength of potential antecedence.

Paul identifies five classes of potential antecedence strength. These classes are:

Class I:	1. The sole referent of a given gender and number (singular or plural) last mentioned within an acceptable distance, **OR**
	2. The *focus* or the head of the *expected focus list* for the previous sentence.
Class II:	The last referent of a given gender and number last mentioned within an acceptable distance.
Class III:	A focal point that filled the same syntactic role in the previous sentence.
Class IV:	1. A referent that has been previously mentioned, **OR**
	2. A referent that is a member of a previously mentioned set that has been mentioned within an acceptable distance.
Class V:	A referent that is known to be a part of a previously mentioned item.

The current focus and the expected focus list can be found by using the algorithm developed and reported by Sidner in [41]. That report specifies a focus algorithm in detail (this algorithm appears in Figure 3-9 ahead), and *Paul* uses it to find the expected focus. The algorithm calls for the ordering of the various noun phrases in a sentence by their syntactic and semantic roles, as well as the order in which they appear in the sentence. As these semantic and syntactic roles are determined, *Paul* creates and modifies the expected focus list as the sentence is being generated.

Distance is the number of clauses between the current one and the one which contains the most recent reference to a specific item. To see why this is important, consider the following example.

```
T1.4 1. John sent a letter to Mary.
     2. Fred found the letter and read it.
     3. He told George about it.
     4. George gave it to Pete.
     5. Pete hid it.
     6. She never got it.
```

The reader identifies that the subject *she* in T1.4-6 is Mary, after a moment's thought. But *she* has to refer to a female, and in all of T1.4 the only female mentioned is Mary. Why, then, can't the reader immediately associate this reference with its antecedent? The answer is distance. There are five clauses between sentences T1.4-1 and T1.4-6. With so many referents introduced between the anaphor *she* and its antecedent Mary, the reader loses track, and cannot make the immediate connection. While the reader is able to eventually trace down the reference in this example, it might not be always possible since 20 or 200 or even 2000 clauses could be between the anaphor and its antecedent.

Paul arbitrarily decides that a distance of two clauses is the maximum acceptable distance for natural anaphor recovery. This is enforced by only keeping the relevant information about the focal points of the last two clauses. The relevant information includes the gender and number of each focal point, as well as their syntactic and semantic roles in the clauses they appear in.

Once the focal points have been classified as to their strength of potential antecedence, it is relatively easy to determine which form of lexical substitution would be acceptable, based on these forms' *strength of antecedence recovery*. The definite noun phrase has the strongest, because it is not really a lexical substitution. It does, however mark the item being referred to as old information, and therefore provides a useful function when no form of lexical substitution is appropriate.

Synonym substitution is also relatively safe in that it too does not generate true anaphora. However, it doesn't by itself distinguish new information from old. Of the two previously discussed tasks for cohesion, that of avoiding needless repetition, and that of marking new information from old, synonym substitution is capable of obtaining only the first goal. Therefore, it is unsuitable as a means in creating anaphora to distinguish previously mentioned items from new ones, and *Paul* doesn't include synonyms in its options for lexical substitution.

That is not to say that synonyms have no place in a text generation system, nor that *Paul* ignores them completely. Synonyms that are members of the same partitioned set are interchangeable. This is not true only in unusual circumstances where there is a need to use an exact word in the text. The decision that a specific word must be used is one of *utterance planning*, not of *utterance realization*. If an item is marked as having to be expressed by a specific word, then *Paul* is capable of generating the text using that specific word. Otherwise, *Paul* randomly selects from the set of equivalent synonyms, thereby achieving variation in the text without fear of incorrectly affecting the intended style and mood.

The next easiest type of lexical substitution to recover is the superordinate substitution. This is true because not all the specific information about the antecedent is lost. Furthermore, because *Paul* insures that all superordinate substitutions will be made unambiguous by adding sufficient modifiers to make the reference unique, recovery from a superordinate substitution is not difficult at all.

Pronouns and general class nouns are the most difficult to recover. Because they provide so little information, they could in general refer to several possible antecedents. The only information directly obtainable from them is gender and number. (And with pronouns, even gender is lost in English if the pronoun is plural.) These forms of lexical substitution have the weakest strength of recovery.

There is an additional problem with general class noun substitution. General class nouns tend to be very informal, extremely personal and familiar, and often derisive and abusive. Obviously, using general class nouns would have a severe impact on the generated text. Controlling the overwhelming effects general class nouns would have on the style and mood of text is beyond the scope of this work. Therefore, while *Paul* can generate general class noun substitutions, unless a text is specifically marked as informal and familiar, a pronoun substitution will be selected.

After a focal point has been found and its class identified, *Paul* has make the appropriate substitution. Deciding which lexical devices can be used on which classes of focal points under which circumstances is a difficult problem. There are issues in addition to achieving understandable cohesion. It is always possible to choose a lexical substitution that has a stronger antecedence recovery than is required, and in fact this is sometimes done by natural speakers. The decision of how to map the various classes of focal points to the lexical substitutions is affected by the desired style of the text to be generated. As the style can change within a text to emphasize something or make a specific idea clearer, this mapping decision must be modified. Unfortunately, such an investigation into changing style and its effects on the selection of lexical substitution is beyond the scope of this work.

Paul makes an arbitrary selection of style in choosing lexical substitution devices. Class I focal points are replaced by pronouns, superordinate substitution is performed on Class II points, and those of Class IV and V become definite noun phrases. Under most circumstances, Class III focal points are subject to superordinate substitution. However, if the previous reference to the item is a Class I focal point, the Class III instance also becomes a pronoun. Intuitively, in order to properly match an element with a lexical substitution to replace it, as the strength of potential antecedence of the element becomes weaker, the strength of antecedence recovery must become stronger.

The significant difference of this work from others is that it addresses the problem of lexical substitution, and cohesion in general, in a methodical manner. Through the use of syntactic and semantic information, the strength of potential antecedence of each focal point is made to determine the minimal features required to generate an understandable reference. A lexical substitution is then selected, based on its strength of antecedence recovery, to provide these minimal features. In this way, the dual tasks of cohesion, the avoiding of repetition and the marking of new information from old, are both achieved.

A few words should be said on the limitations of *Paul*. First, and most obviously, *Paul* is strictly an utterance realization system. There is no provision for utterance planning, and as a complete generation system, *Paul* cannot stand alone. A second limitation is that *Paul* performs only lexical substitution, which is *not* the only cohesive device available in English. Other devices, such as ellipsis and conjunction, have not been investigated to any depth in this work.

Another limitation is that while *Paul* addresses some of the issues of intersentential relationships, these are fairly local issues. There is no attempt to generate text of more than a paragraph at a time. The effects of cohesion, and lexical substitution as a particular device to achieve cohesion, on paragraph structure, and similarly the effects of paragraph structure on cohesion and lexical substitution are topics far beyond the scope of this work. However, work on the paragraph, the level of text generation that *Paul* addresses, could not be seriously attempted until isolated sentence generation had been mostly mastered. It is felt that work on larger texts consisting of many paragraphs cannot be feasibly attempted without first addressing the issues of single paragraph generation.

1.5. Outline of the Remaining Chapters

This chapter has served as a brief introduction to the problem of lexical substitution in computer text generation. The next chapter will provide a detailed discussion of cohesion in English, why it is necessary, and various methods for achieving it. Chapter 3 describes lexical substitution as a cohesive device. In this chapter, we will see what is gained by the inclusion of lexical substitution, as well as what the limitations of such devices are. We will also see in detail how *Paul* incorporates lexical substitution into the generated text. Chapter 4 gives an introduction to NLP, the language *Paul* is written in. Here we will also see the general algorithm used in NLP to generate text. The chapter concludes with a discussion of the generation paradigm used in *Paul*. (Readers interested only in the linguistic results of *Paul* can skip most of this chapter. Except for section 4.11, it is not needed to understand the system's underlying theory nor *Paul's* achievements.) Chapter 5 presents an example text worked out in detail. The output will also be compared to "incorrect" texts, that is, texts without any cohesion and with uncontrolled lexical substitution, in order to graphically illustrate the necessity of controlled lexical substitution. In Chapter 6, current work related to *Paul* will be discussed. And finally, Chapter 7 will conclude this work, describing limitations to the system and future areas of research, as well as presenting the achievements of *Paul.*

2. Cohesion

2.1. Introduction

The purpose of communication is for one person (the speaker or writer) to express her thoughts and ideas so that another (the listener or reader) can understand them. There are many restrictions placed on the realization of these thoughts into language so that the listener may understand. The speaker must organize her ideas and present them in sentences that are complete and grammatical. The sentences must be arranged and realized in such a way that the thoughts naturally progress for the listener in the way that the speaker intended.

One of the most important requirements for an utterance is that it seem to be unified, that it form a *text*. Utterances that are not so unified, that seem to consist of random sentences, are confusing and are usually dismissed as not being serious attempts at communication. Unfortunately, there are no codified rules for what makes an utterance a unified text, the way there is for deciding whether a given sentence is grammatical. While most people have little trouble identifying whether most passages are text or isolated sentences, there are many instances where the answer is not clear. Text is a matter of degree, and what one might be willing to defend as intelligent text, another might insist on branding as a collection of isolated ramblings. However, we are all sensitive to the presence---or lack---of text in an utterance, and we require it in our communications.

The theory of text and what distinguishes it from isolated sentences that is used in *Paul* is that of of Halliday and Hasan [11]. We have already implied that text is not grammatical, and indeed it is not. Sometimes text is seen as a kind of "meta-sentence" following grammatical rules. As a phrase is built from words along strict rules, as a clause is built from phrases, as a sentence is built from clauses, so is a text built from sentences. If this were true, there would be rules governing the order of the sentences and how they appear within the text, but this is not the case [19]. The text is not a grammatical or syntactic unit, it is a *semantic* unit. A text isn't *constructed* with sentences, it is *realized* by them. Therefore, the understanding of text will not be found by investigating their structure.

2.2 The Goals of Cohesion

If this unity found in text is not structural, there must be other factors that provide it. One of the items that enhances this unity is *cohesion*. Cohesion refers to the linguistic phenomena that establish relationships between sentences, thereby tying them together. There are two major goals that are accomplished through cohesion that enhance a passage's quality of text. The first is the obvious desire to avoid unnecessary repetition. A section that referred to an item using the same words with no variety would soon become tedious to read.

The other goal is that new information must be distinguished from old in order that the listener can fully understand what is being said. One reason this is true is that it is necessary to avoid ambiguity. If the speaker refers to an item a second time without clearly marking it as an element that has been previously mentioned, the listener may interpret the reference as one to a completely new item.

```
T2.1 1. The room has a large window.
     2. The room has a window facing east.
     3. The room has a window overlooking the
        backyard.
     4. The room has a window through which
        the sun shines in the morning.
```

How many windows does the room have, four or one? If the room has only one, the speaker of T2.1 would be accused of trying to deceive the listener, although strictly speaking, T2.1 might be completely true. The problem is that the listener will want an indication that the windows referred to in the four sentences are actually all the same window. The way the speaker would provide this indication is through the use of cohesion.

```
T2.2 1. The room has a large window.
     2. It faces east.
     3. It overlooks the backyard.
     4. It is located so that the sun
        shines through it in the morning.
```

2.3. Cohesive Relations

Cohesion is created when the interpretation of an element is dependent on the meaning of another. The element in question cannot be fully understood until the element it is dependent on is identified. The first *presupposes*[11] the second in that it requires for its understanding the existence of the second. As an example, consider the sentence **T2.3.**

```
T2.3: So he did.
```

Of course, by itself out of context, T2.3 is nonsensical. We know someone did something, but we have no idea who that someone was, or what it was he did. The problem is that the sentence has two items, *he* and *did*, that presuppose the existence of previous information. Without this information, the reader cannot understand the sentence.

An element of a sentence presupposes the existence of another when its interpretation requires *reference* to another. In T2.3, *he* refers to the someone we hypothesized, and *did* refers to that person's action. If the sentence had been preceded by "John wanted to buy a kite," we could easily see that *he* now refers to John, and that *did* refers to buying a kite. Once we can trace these references to their sources, we can correctly interpret these elements in T2.3.

The very same devices that create these dependencies for interpretation help distinguish old information from new. If the use of a cohesive element presupposes the existence of another reference of the element for its interpretation, then the listener can be assured that the other reference exists, and that the element in question can be understood as old information. Therefore, the act of associating sentences through reference dependencies helps make the text unambiguous, and cohesion can be seen as a very important part of text.

2.4. Cohesion vs. Coherence

We have seen how cohesion creates dependency relationships between sentences, allowing a passage both to avoid tedious repetitions and to clearly distinguish old information from new, thereby enhancing the quality of text that the passage exhibits. However, we would be very wrong to assume that this is not all that is required for a passage to be considered a text. Consider **T2.4.**

```
T2.4 1. Fred has a green car.
     2. His elephant likes peanuts.
     3. The car has whitewalls.
```

This passage exhibits all the features of cohesion that have been thus far discussed. There are intersentence dependency relationships; *his* in **T2.4-2** and *the car* in **T2.4-3** refer back respectively to *Fred* and *a green car* of sentence **T2.4-1.** There are no unnecessary repetitions; the passage does not say "Fred's elephant" in **T2.4-2** nor "Fred's green car" in **T2.4-3.** And old information is clearly marked; we know the person referred to in **T2.4-2** is the same Fred of **T2.4-1,** and that the car of **T2.4-3** is the same as the one in **T2.4-1.** But one would still be hard pressed to argue that **T2.4** is a unified text.

The reason this is true is that T2.4 lacks *coherence* [15, 16, 17]. While the *interpretations* of the sentences demonstrate the presupposition dependency of cohesion, the *meanings* of the sentences are unrelated, eliminating any sense of text. The distinction here is important. The *interpretation* of sentences can be viewed as understanding sentences individually. Cohesion creates presupposition dependencies so that the understanding of the individual sentence is dependent on the other sentences of the passage. The *meanings* of sentences can be viewed as the understanding of the contents of the sentences as they relate to each other. Coherence involves such factors as relevancy (the factor **T2.4** violates), temporal relationships, and contrasting or parallel relationships. These factors are used to determine which of the myriad facts available should be presented in the discourse, which order they should be presented in, and the manner in which they should be presented. These are exactly the problems of utterance planning, while the problems addressed by cohesion, how to mark old information from new, how to avoid repetitions, and how to link sentences together once their contents are known, are exactly the problems of utterance realization. Therefore, coherence is the phenomenon that enhances the quality of text at the utterance planning stage, while cohesion is the phenomenon that increases the quality of text at the level of utterance realization.

2.5. Cohesive Devices

Several kinds of cohesive devices have been identified [11]. A brief overview of these might prove useful. However, it should be remembered that these classes are not strongly partitioned and that a good deal of overlapping exists. The following discussions will use the classifications defined by [11].

2.5.1 Reference

Perhaps the most general and widely used form of cohesion is that of *reference*. As we have seen, cohesion is created when the interpretation of an element is dependent on another. That is, the information required to understand the current instance of the element must be obtained by retrieving the previous instance. The class of devices known as reference are distinguished from other classes in that the information being retrieved is the actual identity of the current element. The cohesion occurs from the continuity of reference. Reference can be further dividcd into three types, demonstrative, comparative, and personal.

The class of demonstratives is the demonstrative pronouns, *this*, *that*, *here*, *now*, *today*, etc.

```
This is my favorite song.
That is a mean thing to say!
Here is your pen.
Now is the time for all good men to come to the
aid of their country.
Today is the first day if the rest of your life.
```

the general meaning of demonstratives is one of proximity (temporal proximity in the case of *then*, *now*, etc.). *This*, *these here*, *now* imply a nearness, while *that those*, *there*, *then* imply a distance.

Demonstratives tend to be restricted to *situational* [38] or *exophoric* [11] contexts. That is, the demonstrative refers to an item (or location or time) in the physical world, rather to elements specifically mentioned in the text.

```
T2.5: When do you want to go?
      Now!
```

The *now* of T2.5 refers to the moment when the person was speaking, not the present time in which this report is being written or read. If this report is put down for a few days and then picked up again, the actual present time has changed, but the *now* of T2.5 has remained constant. This is what is meant by exophoric reference.

The opposite is *endophoric* reference [11], in which the referent is in the text. Of course, ultimately *all* items refer to the physical world[4]. The words *Fred*, *his elephant*, and *his car* of T2.4 all refer to items in (some) real world. However, they are not exophoric in that one does not have to consider the *situation* of that world to understand the references, as one must do for T2.5. Demonstratives can be used in an endophoric role,

[4]Or at least some hypothetical world. The distinction is irrelevant here.

although it is less common. Generally, they occur when the demonstrative is used to refer to the discourse itself.

```
This is what is meant by endophoric.
```

Comparative references are those of similarity. *Same, identical, equal,* and their adverbial forms are comparatives of identity, *similar, additional,* and their adverbial forms are of similarity, *other, different, else,* are difference, and *better, more, less* and all comparative adjectives and adverbs are for particular comparison. Comparatives are used to express the degree of likeness two items have (or lack). Particular comparatives are used when the similarity with respect to a specific property is to be discussed.

```
That's the same thing I always say.
Other people like it.
New York has more people than Boston.
```

The last kind of reference is the personal reference. This refers to the class of personal pronouns, including subjective, *he, she, it, they,* objective, *her, him, it, them,* possessive, *its, his, hers,* and reflexive, *herself, itself, himself, themselves.* Personal pronouns are used to refer directly to a specific entity, either endophorically or exophorically. While the other types of reference expressed relationships of proximity or similarity, personal reference expresses a relationship of identity. Personal pronouns simply refer to the element in question without additional meaning.

2.5.2. Substitution

Substitution is the replacement of one item in the text with another. The distinction between substitution and reference is subtle, but important. Both reference and substitution require the listener to find another instance of the cohesive item in order to interpret it. The difference is in where that other instance can be. With exophoric reference, we must look at the situational context, in the environment of the speaker. Endophoric reference can be viewed the same way, if we accept the text as a special case of environment [11]. Out of context, a listener cannot tell if a specific usage of reference is exophoric or endophoric. Substitution, on the other hand, can always be resolved within the text.

The three types of substitution are nominal, verbal, and clausal. Nominal substitutes are *one, ones,* and *same.*

```
These kites are expensive, but I want one.
The cherry pops are better than the orange ones.
I'll have the same.
```

Nominal substitutions can be made for only the head nouns [38, 23] of noun phrases. Other elements of the noun phrase, such as modifiers, can be replaced *along with* the head noun, but *not* without it.

```
T2.6  1. Mary has a blue dress with stripes.
     2a. Susie has a red one.
    *2b. Cathy has a red dress with ones.
```

Just as nominal substitutes can replace the head nouns of noun phrases, *verbal* substitutes can replace head verbs of verb phrases. The only verbal substitution in English is *do.*

Who wants this? I do!
Jane likes Wagner, and Vickie *does*, too.

As with the restrictions on nominal substitutions, verbal substitutions can be used only on the head verbs of verb phrases. Modifiers can be replaced only along with the head verb.

*Sam likes to walk the dog, and Anastasia likes to do, too.

Finally, *clausal* substitutions replace whole clauses. In English, the clausal substitutes are *so* and *not.*

George will be late. He told me so.
Will it rain? I hope not.

2.5.3. Ellipsis

Ellipsis, as with the other two types of cohesive devices, creates a presupposition dependency. Rather than replacing an element with some device which conveys less meaning, ellipsis completely eliminates the reference. Actually, this could be thought of as a special case of substitution, one in which the *zero* or *null* element is used to replace the specific referent. However, separating the classes is useful. Substitution uses a variable (of sorts) for its replacement. This variable, while having less information than the actual referent, still contains some, such as number for nominals, and tense for verbs. Ellipsis, on the other hand, by replacing the referent with nothing, offers nothing in the way of information. The proper referent must be identified in order to gain *any* information.

Since ellipsis is a special case of substitution, the two types of ellipsis bear strong parallels to their counterparts in substitution, and the same restrictions that apply to these substitutions apply to ellipsis. Nominal ellipsis allows the deletion of the head noun from a noun phrase.

John went to the store and {*John* ellipted} bought a kite.
I like this story. It's the best {*story* ellipted} I've ever read.

It is important to note that while the head noun is ellipted, it still requires agreement with the verb when in the nominative position.

Phyllis goes to the store and {*Phyllis* ellipted} buys a cake.
Julie and Toni go to the store and {*Julie and Toni* ellipted} buy a cake.

In both of the second clauses of these sentences, the verbs must agree in number with the ellipted subjects.

Verbal ellipsis refers to ellipsis within the verb phrase. Again, the normal restriction is that the head verb of the phrase must be ellipted, and other elements of the verb phrase can be ellipted only *with* the head verb.

```
Who broke this vase? Glenn. {broke this vase ellipted}
```

There are also *elliptical operators* which are used to ellipt a verb. These operators consist of the modals, *can, could, will, would, shall, should, may, might must.*

```
Who will wash the car? I will. {wash the car ellipted}
Have you read this? You should. {read this ellipted}
```

Note that *do* is *not* included in this modal list. This is because *do* does not behave as a modal when used in this context [38, 11, 1].

In addition to allowing the head verb to be ellipted, English allows some of the *operators* of the verb phrase, modals and auxiliaries specifically, to be ellipted.

```
John was laughing and {John ellipted} {was ellipted}
crying at the same time.
Fred should have been singing and Mary {should have been ellipted}
playing the piano when Kirk walked in.
```

2.5.4. Conjunction

Conjunction is the first kind of cohesive device that breaks away from the pattern of replacing some element and creating a presupposition dependency. For this reason, conjunctive elements are not cohesive in themselves, but indirectly. Conjunctions do not replace elements in the text, rather they *connect* them, and this is where the dependencies arise. Since a conjunction spans the gap between two elements of a text, its use creates the dependency that both the element being spanned from and the element being spanned to exist.

It is difficult to cleanly partition the various types of conjunction into distinct sets. Not only are the differences subtle and the sets overlapping, but many words will fall into one category one time and another the next, depending on their usage. However, four general categories for conjunction have been identified. They are *additive, adversative, causal,* and *temporal.*

Additive conjunctions continue thoughts by explicitly linking them, by explicitly stating such a link doesn't exist, or by demonstrating possible alternatives. Simple additives include *and, and also.* Negative additives, those which show that a link doesn't exist, consist of negatives like *not, nor,* etc. Additives can be used for emphasis, *furthermore, in addition, besides,* to de-emphasize, *incidently, by the way,* to express alternatives, such as *or, or else, alternatively,* and many other functions.

Adversative conjunctions link elements in some way that is contrary to expectations or desires. These expectations may come from general knowledge of the real world (so-called "common sense") or from the specific context of the passage. Some example of adversative conjunctions are *yet, though, but* for simple adversatives, *actually, on the other hand, in fact* for contrastives, and *in any case, anyhow, at any rate* for dismissals. T2.7 has several examples showing how adversative conjunctions violate expectations.

```
T2.7 1. It was raining. But we went out, anyway.
     2. We went out, though it was raining.
     3. We usually don't let the rain stop us.
        However, this time we stayed in.
```

T2.7-1 and T2.7-2 demonstrate the violation of "common sense" expectations. We expect people to be intelligent enough to stay out of the rain. T2.7-3 violates expectations created by the previous two sentences. After T2.7-1 and T2.7-2, the listener expects the speaker and her group to be people who frequently go out in the rain. This is confirmed by the first sentence of T2.7-3, but this situational expectation is then violated by the second sentence of T2.7-3.

Causal conjunctions express a causal relationship between elements. As with other forms of conjunction, causal conjunctions serve many functions. They can be used to state a forward flow of causality with words like *so, then, hence, consequently.* A reversed causal flow, where the second element is the cause of the first, is possible, *for, because, it follows from* being examples. Conditional causality makes use of *then, in that case, in such an event* and others.

Finally, temporal conjunctions explicitly state the time sequence of tow elements. This temporal flow can be sequential, *then, next, after that,* preceding, *previously, before that,* or simultaneous, *just then, at once,* interrupted, *soon, after a time,* to name some of the possibilities.

2.5.5. Lexical Substitution

Lexical substitution is the final category of cohesive devices. Lexical substitution achieves cohesion through the proper selection of vocabulary, rather than through grammatical constructions, as did the previous cohesive devices. Cohesion is not created through *reference,* as it was with reference, substitution, and ellipsis, nor through *expressing links,* as it was with conjunction, but through *repetition.* Chapter 3 discusses lexical substitution at length, describing the various kinds of lexical substitution, and how they were implemented in *Paul.*

3. Lexical Substitution

3.1. Reiteration

With the exception of conjunction, all the cohesive devices we have looked at so far involve multiple references to the same item. Reference, substitution, and ellipsis replace these references with specific "variables" or "place holders" such as pronouns, or in the case of ellipsis, empty strings. The proper selection of these variables is based on grammatical rules, and not on semantic information concerning the items the variables are replacing. For instance, in choosing the correct personal pronoun, all we need to know is the gender, number, and case of the item to be replaced. We do not need pragmatic information, such as the general class to which the item in question belongs, what other kinds of things are similar to the item in question, or how is the item in question used. Nor do we need semantic information, how does the speaker or the listener feel about this specific item, what overall role is the item playing in the text, what is its current role in this sentence.

Lexical substitution, on the other hand, makes use of pragmatic and semantic information to correctly choose a replacement for the item. That is, rather than *grammatically* replacing an item to achieve cohesion, lexical substitution *lexically* replaces the item [11]. We can call this lexical replacement *reiteration* [11].

Because the selection within grammatical cohesive devices is dictated by the grammar, there is no difficult decision process involved. This is unfortunately not true in the case of lexical substitution. The options are much more varied, and the decision process is consequently more difficult. An example will help demonstrate exactly what these options are.

```
                              *VEHICLE*

        *WATER-VEHICLE*       *LAND-VEHICLE*  *AIR-VEHICLE*

      *SHIP*    *SUBMARINE*    *CAR*  *TRUCK*     *PLANE*

    BOAT   SHIP   SUBMARINE   CAR AUTO   TRUCK       PLANE

LEAKIN' LENA
```

Figure 3-1: Fragment of a Semantic Hierarchy

Figure 3-1 shows a fragment of a possible semantic hierarchy. Let us assume that it is desired to make a reference to the item *BOAT* If we want to use lexical substitution, we must find some semantic replacement for *BOAT*. Given that our semantic structure is a two-dimensional hierarchical tree, we have several options in how to move through the tree to find a suitable replacement. The first, and obviously easiest, way is to not move at all, but stay at the node in the tree for *BOAT*. Another is to move *across* to a sibling node, in this case to *SHIP*. A third is to move to *up* the hierarchy to a parent node. The immediate parent of *BOAT* is **SHIP**, but we are not ruling out moving further up the hierarchy (at least for now), so we can also include **WATER-VEHICLE** and **VEHICLE** We can move *down* the hierarchy to a child node, *LEAKIN' LENA* in our example. Finally, we can move *out* of the hierarchy altogether, using some variable to mark the fact that we've left the plane.

In fact, these very moves through the hierarchical structure are what lexical substitution performs. *Synonymous substitution* moves across the hierarchy to a sibling. *Superordinate substitution* moves up the hierarchy to an ancestor. *General nouns* and *personal pronouns* move us out of the plane of the hierarchy. And definite noun phrases keep us at the same node. The next few sections will examine the various types of lexical substitution, how they move through the hierarchy, and how *Paul* incorporates them into the generated text.

3.2. Synonyms

Synonymous substitution is the replacement of an item with another that has the same meaning. This corresponds to the lateral movement across a hierarchy to a sibling. But not all the siblings in the tree in Figure 3-1 are synonyms. For instance, **SHIP** and **SUBMARINE** are clearly not synonymous. In *Paul*, the semantic hierarchy can be divided into two levels, a *conceptual* level and a *lexical* level. Nodes in the conceptual level represent concepts in the abstract, modeled after the so-called primitive actions [39] and primitive objects [25]. In Figure 3-1, entries in the conceptual level are marked with asterisks. Entries in the lexical level represent actual words in English. These are the words that can be used for the output. In Figure 3-1, these are the entries without asterisks. Only siblings of nodes in the lexical level are synonyms. Therefore, *SHIP* and *BOAT* are synonyms, but **SHIP** and **SUBMARINE** are not.

Obtaining siblings in a tree is a fairly straightforward task, and mechanically generating synonyms presents no problems. However, that does not mean incorporating synonyms into a text generation system is a trivial task. The difficulty comes in the fact that true synonyms may not actually *exist* at all. Two words rarely mean the exact same thing in every context. Even when the literal meanings are identical, words can convey different *moods* and *connotations*. In addition to their meanings, words frequently have associated with them a sense of "goodness" or "badness," "pleasantness" or "unpleasantness." This is what is meant by connotation.

As an example, if one were to look up "odor" in a thesaurus, one might find the entries in Figure 3-2. If a system tried to use these words interchangeably, ignoring their connotations, the sentences of Figure 3-3 could be erroneously generated as equivalent. The problem is that while all the words of Figure 3-2 have the same general meaning, they clearly have different connotations. One possible classification of these words by connotation is shown in Figure 3-4.

Odor
aroma
emanation
foulness
fragrance
odor
scent
smell
stench
stink

Figure 3-2: Synonyms for Odor

The *aroma* of her perfume filled the room.
The *emanation* of her perfume filled the room.
The *foulness* of her perfume filled the room.
The *fragrance* of her perfume filled the room.
The *odor* of her perfume filled the room.
The *scent* of her perfume filled the room.
The *smell* of her perfume filled the room.
The *stench* of her perfume filled the room.
The *stink* of her perfume filled the room.

Figure 3-3: Example of Uncontrolled Synonym Substitution

POSITIVE	NEGATIVE	NEUTRAL
aroma	foulness	odor
emanation	stench	scent
fragrance	stink	smell

Figure 3-4: Classification of Synonyms by Connotation

If the lexical dictionary is arranged in such a way that synonyms are partitioned into distinct sets based on their connotative qualities, then simple synonym substitution is possible. The decision to use a word with a particular connotation is one of utterance planning, while the specific choice is one of utterance realization. *Paul* performs this selection randomly from within the proper set of synonyms. Since one of the purposes of synonym substitution in the first place is to avoid unnecessary repetition, the random selection process uses a global memory variable to "remember" the words it has already selected. Given a list of words to randomly choose from, the system will not repeat itself unless every item on the list has already been used. If the entire list has previously appeared, then every member of the list is "forgotten" by being removed from the global memory variable, and the whole process is begun again.

While we've been mostly discussing cohesive devices as they apply to substitutions for nouns, most of these devices can also be used for verbs. This is especially true for synonyms. By setting up a hierarchy of primitive actions [39], *Paul* can choose from the correct list of verbs that mean the desired action that have the required connotations. The same mechanisms used for selecting synonymous noun substitutions are used for selecting synonymous verb substitutions. In this way, *Paul* achieves a great deal of variety in the text it generates without creating sentences with erroneous connotations.

3.3. Superordinates

Superordinate substitution is the replacement of an element with a noun or phrase that is a more general term for the element. For instance, in Figure 3-1, the superordinate of *LEAKIN' LENA* is *BOAT*, that of *BOAT* is **SHIP**, and again for **SHIP** the superordinate is **WATER-VEHICLE**. Finally, the superordinate for **WATER-VEHICLE** is **VEHICLE**. Superordinates can continue for as long as the hierarchical tree will support.

As with synonymous substitution, the mechanics for performing superordinate substitution is fairly easy. All one needs to do is to create a list of superordinates by tracing up the hierarchical tree, and randomly choosing from this list. However, there are several issues that must be addressed to prevent superordinate substitution from being ambiguous or making erroneous connotations. The erroneous connotations occur if the list of superordinates is allowed to extend too long. An example will make this clear. Let us assume that we have a hierarchy in which there is an entry *FRED*. The superordinate of *FRED* is *MAN*, for *MAN* *HUMAN*, *ANIMAL* for *HUMAN*, and *THING* for *ANIMAL*. Therefore, the superordinate list for *FRED* is *(MAN HUMAN ANIMAL THING)*. While referring to Fred as ***the man*** seems fine, calling him ***the human*** seems a little strange. And furthermore, using ***the animal*** or ***the thing*** to refer to Fred is actually insulting.

The reason these superordinates have negative connotations, even though Fred *is* of course an animal and a thing, is that there are essential qualities that humans possess that separate us from other animals. Calling Fred "animal" implies that he lacks these qualities, and is therefore insulting. The reason "human" sounds strange is that it is the highest entry in the semantic hierarchy that exhibits these qualities. Talking about "the human" gives one the feeling that there are other creatures in the discourse that are not human.

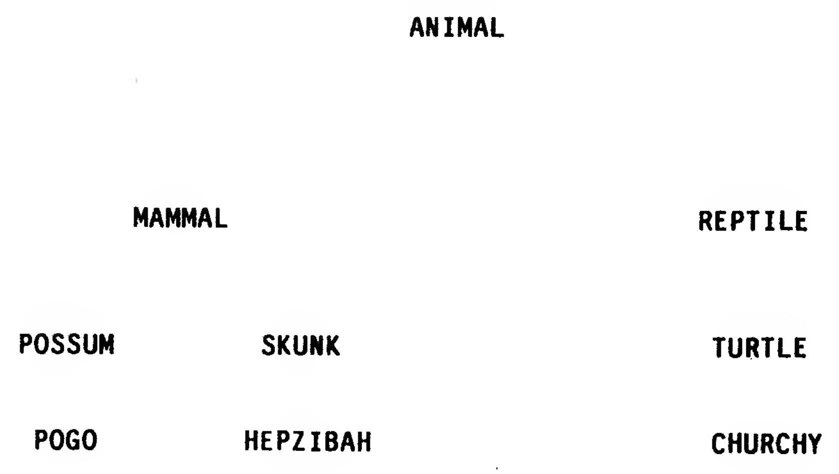
Paul is sensitive to the connotations that are possible through superordinate substitution. The essential quality identified for superordinate substitution is intelligence. The system first sees if the item to be replaced with a superordinate substitution is intelligent, either directly or by semantic inheritance. If so, a superordinate list is made only of those entries that have themselves the quality of intelligence, again either directly or through inheritance. If the item to be replaced *doesn't* have intelligence, the list is allowed to extend as far as the hierarchical entries will allow. Once the proper list of superordinates is established, *Paul* randomly chooses one, preventing repetition the same way it did in the random selection of synonyms.

The other problem of superordinate substitution is that it may introduce ambiguity. Consider the semantic hierarchy of Figure 3-5. If we wanted to perform a superordinate substitution for *POGO,* we would have the superordinate list *(POSSUM MAMMAL ANIMAL)* to choose from. But *HEPZIBAH* is also a mammal, so *the mammal* could refer to either *POGO* or *HEPZIBAH.* And not only are both *POGO* and *HEPZIBAH* animals, but so is *CHURCHY,* so *the animal* could be any one of them. Therefore, saying *the mammal* or *the animal* would form an ambiguous reference which the listener or reader would have no way to understand.

Paul recognizes this ambiguity. Once the superordinate has been selected, *Paul* tests it against all the other nouns mentioned so far in the text. If any other noun is a member of the superordinate set in question, if the superordinate is an ancestor to any of the other nouns, the reference is ambiguous. However, by using a feature of the element to be replaced as a *modifier,* the reference can be disambiguated. For instance, Figure 3-5 tells us that possums are grey, and that *POGO* is a possum. Additionally, neither *HEPZIBAH* nor *CHURCHY* are grey. Therefore, while *the mammal* and *the animal* are ambiguous, *the grey mammal* and *the grey animal* are not. If the superordinate selection proves *not* to be ambiguous, such as if *POSSUM* were to be chosen in this example, a disambiguating modifier is not necessary, and none is chosen.

The features that *Paul* recognizes for disambiguating superordinates in Pogo world are gender, size, color, and skin type (furry, scaled, feathered). As with synonym selection and superordinate selection, choice of the disambiguating feature is random, using the same function to prevent repetition of a feature until the entire list has been exhausted. Once the feature is selected, the proper value of the feature for this element is found through inheritance.

However, there is the further complication that the disambiguating modifier doesn't disambiguate. Since the feature is selected randomly, the one for our example could have been skin type. *The furry animal* is little better than *the animal* because both *POGO* and *HEPZIBAH* are furry, both being mammals. And *the furry mammal* is uselessly redundant because *all* mammals are furry in this world. Similarly, if size had been the feature selected, the results would have been either *the small mammal* or the *small animal,* and again the



1. POGO IS A MALE POSSUM.

2. HEPZIBAH IS A FEMALE SKUNK.

3. CHURCHY IS A MALE TURTLE.

4. POSSUMS ARE SMALL, GREY MAMMALS.

5. SKUNKS ARE SMALL, BLACK MAMMALS.

6. TURTLES ARE SMALL, GREEN REPTILES.

7. MAMMALS ARE FURRY ANIMALS.

8. REPTILES ARE SCALED ANIMALS.

Figure 3-5: Another Sample Hierarchy

phrase is as ambiguous as if no modification had occurred.

Paul avoids this problem by testing the selected modifier. When the chosen superordinate is found to be ambiguous, a list is made of all the problem nouns that it could refer to. After the disambiguating feature is selected and the proper value determined, this value is checked against the values each of the problem nouns on this list would inherit for the feature. If any one of the problem nouns inherits the same value for the feature, the feature is rejected, and a different one is randomly selected. This process continues until a feature is found which truly disambiguates the superordinate reference.

3.4. General Nouns

General nouns are the first kind of lexical substitution that move us out of the hierarchy plane. That is, rather than attempt to find a node in the hierarchical tree that can be used as a substitute for the element in question, general nouns serve as "tokens" that replace the element.

General nouns consist of those nouns that can be used to replace the major noun classes. *People, person, man, woman, child, boy, girl* are examples of general nouns for the human class. *Creature, beast* are non-human animate general nouns. For inanimate concrete count nouns, we have *thing, object,* while for inanimate concrete mass nouns we have *stuff.* Inanimate abstract nouns can be replaced by *business affair, matter.* Nouns representing actions have *move* for a general noun, while nouns of location have *place, spot.* Finally fact nouns use *question, idea* for general nouns.

```
Tom doesn't look well. The old boy must be sick.
I just danced with Grandmother. The dear girl still has it.
I just love Paris. This place is so alive.
```

These are very close to superordinates, and in fact originally derive from them. But they are not identical. Superordinates are used only when the element to be replaced is an actual member of the superordinate set. General nouns are not as strict in that close approximations of the proper superordinate are allowed. In the second pair of sentences of the above example, the speaker is not really stating that Grandmother is a female child. Additionally, general nouns tend to have "empty" modifiers, adjectives that are not meant to be taken literally. In the first pair of sentences above, we are not being told that Tom is old, and it would be a mistake to assume so.

This idea of using "tokens" or "variables" to replace elements of a sentence is very similar to the grammatical cohesive device of reference. With both, cohesion is formed because the interpretation of the element is dependent on the successful retrieval of another element. The difference is in the type of "variable" that is used. The selection of reference substitutes is purely grammatical. If one wants a personal pronoun for Fred to serve in a subjective position, *he* must be used. General nouns, on the other hand, derive from the superordinates hierarchy.

Another important difference between reference substitutes and general nouns is that general nouns have connotations that reference substitutes do not. General nouns, especially those for the human class, give a strong impression of familiarity. In business correspondence, one would probably not want to refer to a client as "the old boy" or "the dear girl." Additionally, general nouns can be used epithetically to be insulting, whereas reference substitutions are semantically neutral. (Most expletives can be used as general nouns in this way.)

3.5. Personal Pronouns

As do general nouns, personal pronouns represent movement out of the plane of the hierarchy by using a "variable" to replace the element. Strictly speaking, personal pronouns are not a device of lexical substitution. They belong to the grammatical cohesion device of reference substitution. However, there were several reasons for including pronouns in *Paul*. The first one, as explained above, is that reference substitution is very close to general noun substitution, and the incorporation of general nouns while excluding personal pronouns almost seems arbitrary. The second is that personal pronouns are probably the most widely used of any of the cohesive devices used in English. Any attempt to approach natural text without the use of pronominalization is almost doomed before it begins. For these reasons, *Paul* incorporates personal pronouns in its lexical substitution devices.

Because the selection of the personal pronoun is strictly grammatical, the mechanism to perform this task is very straightforward. Once the syntactic case, the gender, and the number of the element in question are determined, the correct pronoun is dictated by the language.

3.6. Definite Noun Phrases

The final lexical substitution available in *Paul* is the definite noun phrase. A definite noun phrase is simply created using a definite article, *the* in English, as opposed to an indefinite article, *a* or *some*. Of course, definite articles are used with the other types of lexical substitutions, but they can also be used with a repetition of the *exact same word* for the element. This represents not moving at all in the hierarchy. In its simplest form, the definite article refers to a specific known element. The way in which it is known can vary. It could be exophoric, as in "the man over there," or endophoric, as in "I had a balloon, but the balloon broke." When used endophorically, the definite article clearly marks an item as one that has been previously mentioned, and is therefore old information. The indefinite article similarly marks an item as *not* having been previously mentioned, and therefore being new information. Because English has only one definite article, *the*, the mechanism for definite article selection is not an issue.

The capacity of the definite article to mark an element as old information makes its use required with superordinates and general nouns.

```
My sheepdog is smart. The dog fetches my newspaper every day.
*My sheepdog is smart. A dog fetches my newspaper every day.

George worries me. The poor boy works too hard.
*George worries me. A poor boy works too hard.
```

3.7. Controlling Lexical Substitution

While the mechanisms for performing the various lexical substitutions are conceptually straightforward, they do solve the entire problem of using lexical substitution. So far, we've only discussed how to use these cohesive devices *once they've been selected.* Nothing was said about how the system chooses which cohesive device to use. This is a serious issue in that lexical substitution devices are *not* interchangeable. Consider the story in Figure 3-6. This story is unintelligible, and of course unacceptable as output for computer generated text. The problem is that the cohesive devices were chosen randomly. If the selection of lexical substitution devices is not carefully controlled, the resulting passage will not be understandable, and certainly will not be acceptable text.

```
HE CARES FOR THE WOMAN. BETTY LIKES THE POLICEMAN, TOO. THE OLD BOY
GIVES ONE TO HER. THE NURSE LIKES THE RING.
```

Figure 3-6: Story with Uncontrolled Lexical Substitution

The reason why indiscriminately chosen lexical substitutions make a passage unintelligible is that lexical substitutions, as do most cohesive devices, create text by using *presupposed dependencies* for their interpretations, as we have seen. If those presupposed elements do not exist, or if it is not possible to correctly identify which of the many possible elements is the one presupposed, then it is impossible to correctly interpret the cohesive element, and the only possible result is confusion. A computer text generation system that incorporates lexical substitution in its output must insure that the presupposed element exists, and that it can be readily identified by the reader.

Paul controls the selection of lexical substitution devices by conceptually dividing the problem into two tasks. The first is to identify the *strength of antecedence recovery* of the lexical substitution devices. The second is to identify the *strength of potential antecedence* of each element in the passage, and determine which if any lexical substitution would be appropriate.

3.7.1 Strength of Antecedence Recovery

Each time a cohesive device is used, a presupposition dependency is created. In order to correctly interpret the element, the item that is being presupposed must be correctly identified. The relative ease with which one can recover this presupposed item from the cohesive element is called the *strength of antecedence recovery.* The stronger an element's strength of antecedence recovery, the easier it is to identify the presupposed element.

The lexical substitution with the highest strength of antecedence recovery is the definite noun. This is because the element is actually a repetition of the original item, with a definite article to mark the fact that it is old information. There is no real need to refer to the presupposed element, since all the information is being repeated.

The next highest is the synonym. Since properly partitioned synonyms are semantically equivalent, they can be treated as an extension of the repetition that occurs with the definite noun phrase. When used by themselves, synonyms do not create the presupposition dependency that ties sentences together. Therefore synonyms are not used by *Paul* to achieve cohesion between sentences. They *are* used to prevent repetition, but this task is independent of the intersentential cohesion being controlled here. Therefore, synonymous substitution is allowed to occur freely whenever possible.

Superordinate substitution is the lexical substitution device with the next highest strength of antecedence recovery. Presupposition dependency does genuinely exist with the use of superordinates, because some information is lost. When we move up the semantic hierarchy, all the traits that are specific to the element in question are lost. The higher up we go, the more information is lost. To recover this, and fully interpret the reference at hand, we must trace back to the original element in the hierarchy. Fortunately, the manner in which *Paul* performs superordinate substitution facilitates this recovery. By insuring that the superordinate substitution will never be ambiguous, the system only generates superordinate substitutions that are readily recoverable.

The lexical substitution device with the next strength of antecedence recovery is the general noun. These items provide almost no information. Since they move us out of the plane of the semantic hierarchy, general nouns serve as little more than place holders for elements in the sentence. As we have seen, general nouns have a large impact on the style of a passage, making it much more familiar and informal, and possibly adding a derisive tone to the text. Since such considerations of style are beyond the scope of this thesis, *Paul* has been designed to not choose general nouns as a possible lexical substitution, although the mechanism for generating general nouns has been incorporated into the program.

The final cohesion device used by *Paul*, personal pronouns, has the lowest strength of antecedence recovery. Pronouns genuinely are nothing more than place holders, variables that maintain the positions of the elements they're replacing. A pronoun contains absolutely no semantic information, only syntactic. The only readily available pieces of information from a pronoun are the syntactic role in the current sentence, the gender, and the number of the replaced item. For this reason, pronouns are the hardest to recover of the substitutions discussed.

3.7.2. Strength of Potential Antecedence

While the forms of lexical substitution provide clues (to various degrees) that aid the reader in recovering the presupposed element, the actual way in which the element is currently being used, how it was previously used, its circumstances within the current sentence and within the entire text, can provide additional clues. These factors combine to give the specific reference a *strength of potential antecedence.* Some elements, by the nature of their current and previous usage, will be easier to recover independent of the lexical substitution device selected.

Strength of potential antecedence involves several factors. The *syntactic role* the element is playing in the current sentence, as well as the previous reference, the *distance* of the previous reference from the current, and the current focus of the text all affect an element's potential strength of antecedence. *Paul* identifies five classes of potential antecedence strength, Class I being the strongest and Class V the weakest, as well as a sixth "non-class" for elements being mentioned for the first time. These five classes are shown in Figure 3-7.

Class I:	1. The sole referent of a given gender and number (singular or plural) last mentioned within an acceptable distance, **OR**
	2. The *focus* or the head of the *expected focus list* for the previous sentence.
Class II:	The last referent of a given gender and number last mentioned within an acceptable distance.
Class III:	A focal point that filled the same syntactic role in the previous sentence.
Class IV:	1. A referent that has been previously mentioned, **OR**
	2. A referent that is a member of a previously mentioned set that has been mentioned within an acceptable distance.
Class V:	A referent that is known to be a part of a previously mentioned item.

Figure 3-7: The Five Classes of Potential Antecedence

Once an element's class of potential antecedence is identified, the selection of the proper lexical substitution device is easy. The stronger an element's potential antecedence, the weaker the antecedence recovery of the lexical substitution. Therefore, Class I elements, those with the highest strength of potential antecedence, are replaced with personal pronouns, the substitution with the lowest strength of antecedence recovery. Class II elements, with the next highest strength of potential antecedence, are replaced with superordinates, the next lowest cohesive device. Class III elements are unusual in that the device used to replace them can vary. If the previous instance of the element was of Class I, if it was replaced with a pronoun, then the current instance is replaced with a pronoun, too. Otherwise, Class III elements are replaced with superordinates, the same as Class II. Class IV and Class V elements are both replaced with definite noun phrases. These mappings from potential antecedence classes to lexical substitution devices is illustrated in Figure 3-8.

Class I	Pronoun Substitution
Class II	Superordinate Substitution
Class III (previous reference Class I)	Pronoun Substitution
Class III	Superordinate Substitution
Class IV	Definite Noun Phrase
Class V	Definite Noun Phrase

Figure 3-8:
Mapping of Potential Antecedence Classes to Lexical Substitutions

The decision on which lexical substitutions would be used to replace which potential antecedence classes was made fairly arbitrarily. This mapping intuitively makes sense. As the strength of potential antecedence gets weaker by class, the strength of antecedence recovery gets stronger with the associated lexical substitution. However, there is no formal justification to this exact mapping. The choice of which lexical substitution to use for an element, once that element's class has been identified, is a question of style. There is usually more than one type of lexical substitution that will serve the goals of cohesion. The difference between them is that they will have different impacts on the style and mood, the "feeling ," of the text.

```
T3.1 1. Hank lost Robin's book.
     2a. She was heartbroken.
     2b. The girl was heartbroken.
     2c. The poor girl was heartbroken.
```

Each of the responses of T3.1-2 are acceptable following T3.1-1, but they have different impacts on the overall style of T3.1. T3.1-2a has a more informal, conversational tone, while T3.1-2b is more formal. And T3.1-2c is very informal, and implies sympathy on the speaker's part. As was stated above, the investigation of style and its impact on lexical substitution selection and vice versa is beyond the scope of this report. Therefore, an arbitrary style was chosen for *Paul*, as reflected in Figure 3-8.

3.7.3. Focus

One of the most important factors used in determining the potential antecedence class of an element is *focus* [41, 35, 15, 16, 17]. Focus is what a discourse is about [38]. It is the central idea around which the sentence revolves.

In order to identify the current focus or expected focus list, *Paul* uses the detailed algorithm for focus developed by Sidner [41]. Figure 3-9 shows this algorithm.

```
Choose an expected focus as:
    The subject of a sentence if the sentence is an is-a or a
    there-insertion sentence.

    The first element of the default expected focus list, computed from
    the semantic case relations of the verb as follows:
        Order the set of phrases in the sentences using the following
        preference schema:
            affected case unless the affected case is a verb complement in
            which case the affected case from the complement is used

            all other semantic case positions with the agent last

            the verb phrase
```

Figure 3-9: Expected Focus Algorithm

3.7.4. Distance

Another important factor in determining an element's class is *distance*. By this we mean the distance between the current reference and the most recent previous reference for the same item. Distance affects our ability to recover the antecedent for a lexical substitution. As the distance between the referent and its antecedent increase, the number of possible referents is likely to increase, thus making the recovery a confusing process. Additionally, as the distance increases, other elements are introduced and discussed. The focus of these intermediate sentences is obviously not on the element in question. When this element is finally brought back to the reader's attention, it has to be re-introduced as something pertinent to the discussion. Perhaps an example would help make this clear.

```
T3.2 1. John sent a letter to Mary.
     2. Fred found the letter and read it.
     3. He told George about it.
     4. George gave it to Pete.
     5. Pete hid it.
     6. She never got it.
```

The *she* in T3.2-6 must be Mary, since Mary is the only female mentioned in all of T3.2. However, there are five clauses between the initial reference of Mary in T3.2-1 and the pronoun in T3.2-6. With five sentences consisting of six clauses, all of which have the letter as their focus, it seems strange to use the cohesive device with the weakest antecedence recovery to refer to an element that was mentioned in passing (since Mary is *not* the focus of T3.2-1) six clauses ago.

On the other hand, not allowing *any* distance is too restrictive.

```
T3.3 1. John sent a letter to Mary.
     2. The postman lost it.
     3. She never got it.
```

Using the pronoun *she* in T3.3-3 seems perfectly natural and acceptable, even though the sentence it is in, T3.3-3, does not immediately follow the sentence in which the first reference occurred, T3.3-1. There must be some range in distance for which such pronominalization is acceptable, and beyond which it is not. Unfortunately, linguists have not been able to determine the exact scope of this range. It seems that rather than there being an exact cutoff line, there is a continuum of acceptability, as there is with most linguistic features. An additional complication is that this continuum may be able to shift, extending the accepted range for some contexts, and decreasing it for others. Unfortunately, investigation into this linguistic issue is beyond the scope of this report. In *Paul* the acceptable distance was arbitrarily set at two clauses.

3.7.5. Endophoric Limitations

A limitation of this use of focus and distance is that it assumes endophoric references. The possibility of shifting focus by simply gesturing at an object, the definition of distance based on an object's *physical distance* to a referent, rather than its distance in the text, have been ignored. To appreciate the significance of this, consider the following as the first sentence of an instruction manual.

First, loosen the top screw on the carburetor.

Neither *screw* nor *carburetor* have been mentioned before in the text, yet both are presented as definite noun phrases. This is correct because the references are meant to be exophoric, not endophoric. It is assumed that the reader of this sentence has the proper engine in front of her, and can readily identify the carburetor and its top screw by sight. Since *Paul* assumes endophoric references, it would have incorrectly generated this sentence.

First, loosen a top screw on a carburetor.

This one says to loosen *any* screw found on *any* carburetor, and implies that there are more than one of each, a much different message from the first. While the definition of potential antecedence classes used in *Paul* is adequate for strictly endophoric contexts such as children's stories, it would have to be greatly modified before exophoric contexts could be properly generated.

3.8. Comparison with Another System

With all the elaborate mechanisms developed for *Paul*, and their theoretical justifications, as we have been discussing, it may be difficult to judge just exactly what is gained by their inclusion. Therefore, this chapter concludes with an example of a story generated by *Paul* and the same story as it would have been generated with a much simpler algorithm for pronominalization. Figure 3-10 shows the sample story with no form of lexical cohesion. (Figure 3-5 contained the semantic hierarchy for this world.)

POGO CARES FOR HEPZIBAH. CHURCHY LIKES HEPZIBAH, TOO. POGO GIVES A ROSE TO HEPZIBAH, WHICH PLEASES HEPZIBAH. HEPZIBAH DOES NOT WANT CHURCHY'S ROSE. CHURCHY IS JEALOUS. CHURCHY PUNCHES POGO. CHURCHY GIVES A ROSE TO HEPZIBAH. PETALS DROP OFF. THIS UPSETS HEPZIBAH. HEPZIBAH CRIES.

Figure 3-10: The Sample Story

The simple pronominalization rule that will be compared with *Paul* is one that appeared in [22], and is presented here in Figure 3-11. The rule only allows pronominalization if the last reference to the element was in the last sentence. (In other words, this rule uses a maximum acceptable distance of one sentence.) The *previous-pronouns-list* refers to the pronouns that would be used to replace the nouns of the previous sentence. For instance, if the previous sentence were "Both Fred and George like Mary," the previous noun list would be *(Fred George Mary)* and the previous-pronouns-list would be *(he he she)*.

Pronominalization Rule: A repetitive noun phrase in the *current* sentence is replaced by its pronoun only if the pronoun is unique in the *previous-pronouns-list* (that is, no other noun phrases in the *previous* sentence has the same pronoun).

Figure 3-11: The Simple Pronominalization Rule

Let's see what the pronominalization rule of Figure 3-11 would do with the sample story of Figure 3-10. With the first sentence, the previous nouns list is empty, as well as the previous-pronouns-list, and no pronominalization occurs. However, with the second sentence, the previous nouns list is *(Pogo Hepzibah)* and the previous-pronouns-list is *(he she)*. Since *Churchy* of the second sentence is male, the pronoun for Churchy is *he*. With another *he* on the previous-pronouns-list, pronominalization here is blocked. However, *Hepzibah* has the only *she* on the previous-nouns-list, and the final sentence is *Churchy likes her, too.* With the next sentence, *Pogo* cannot be pronominalized because he is not on the list of previous nouns, and even if he were, *Churchy* has a *he* on the previous-pronouns-list, and no pronominalization would occur. *Hepzibah* is still on the list of previous nouns, and is still the only *she* on the previous-pronouns-list, and the resulting sentence is *Pogo gives a rose to her, which pleases her.* Similarly, the other sentences would be processed, and the final story is in Figure 3-12.

POGO CARES FOR HEPZIBAH. CHURCHY LIKES HER, TOO. POGO GIVES
A ROSE TO HER, WHICH PLEASES HER. SHE DOES NOT WANT CHURCHY'S
ROSE. HE IS JEALOUS. HE PUNCHES POGO. CHURCHY GIVES A ROSE TO
HEPZIBAH. PETALS DROP OFF. THIS UPSETS HEPZIBAH. SHE CRIES.

Figure 3-12: Results of Simple Pronominalization Rule

The sample story as generated by *Paul* is in Figure 3-13. (The details of the generation of this story are discussed at length in Chapter 5.)

POGO CARES FOR HEPZIBAH. CHURCHY LIKES HER, TOO. POGO GIVES A
ROSE TO HER, WHICH PLEASES HER. SHE DOES NOT WANT CHURCHY'S ROSE.
HE IS JEALOUS. HE PUNCHES POGO. HE GIVES A ROSE TO HEPZIBAH.
THE PETALS DROP OFF. THIS UPSETS HER. SHE CRIES.

Figure 3-13: *Paul's* Version of Sample Story

The differences between the two algorithms do not manifest until the seventh sentence, *Churchy gives a rose to Hepzibah.* Because the sixth sentence mentions both *Churchy* and *Pogo,* the previous-pronouns-list during the seventh sentence is *(he he),* and the algorithm does not allow *Churchy* to be pronominalized in this sentence. With *Paul,* though, *Churchy* in the seventh sentence is Class III because the referent repeats the syntactic role it had in the previous sentence, in this case subject. When the previous reference was realized as a pronoun, Class III referents are also realized by pronouns, and the resulting sentence is *He gives a rose to Hepzibah.*

The next difference is in the ninth sentence[5]. Because *Hepzibah* wasn't mentioned in the eighth sentence, and the simple pronominalization rule only allows a distance of one sentence for pronominalization, the element is left untouched. *Paul,* on the other hand, uses a distance of two, and the referent is replaced with the appropriate pronoun in *Paul's* version.

This brief example shows that *Paul* is much richer in creating pronominalization than the simple rule of Figure 3-11. And of course, providing other forms of lexical substitution and carefully controlling their use allows *Paul* to generate a large variety of quite natural text. Appendix III contains several examples of actual texts generated by *Paul.*

[5]Of course, the eighth sentence is also different in the two versions. But this difference is because *Paul* identifies *parts* of previously mentioned elements, and classifies them as Class V. Since this is independent of pronominalization and the rule we are contrasting against *Paul* is one for only pronominalization, a comparison for the eighth sentence wouldn't be fair.

4. NLP

4.1. Introduction

This chapter presents an introduction to NLP (for *Natural Language Processor*) as it was implemented for *Paul*. The reader is introduced to the major constructs of the language, and the syntax and semantics of those constructs. A working knowledge of LISP is required to gain a complete comprehension of the presented material, but programming expertise is not necessary---and the reader will certainly not be asked to trace through lines of code. Additionally, after the description the algorithm used in NLP to generate sentences will be discussed, and an example will be provided.

It should be understood that this chapter is *not* intended to serve as a manual or users' guide to NLP, but simply an introduction to some of the concepts central to the language's use. Furthermore, opinions expressed in this chapter are solely this author's and do not necessarily have the agreement or the approval of my colleagues, nor of George Heidorn, the creator of NLP.

NLP is a language created by George Heidorn specifically for natural language processing. The language allows the user to write and execute production rules on frame-like data structures which Heidorn calls records. Since Heidorn's original version of NLP as reported in 1972 [13] was supported by a FORTRAN program, it reflected many of the constraints and special properties of a numerical computational language. By using LISP, a subset of NLP was implemented---essentially the instructions necessary for language generation---without the artificial numerical orientation of Heidorn's version. Consequently, the current version of NLP used for *Paul* is not completely compatible with Heidorn's, and the following descriptions of NLP, while agreeing with Heidorn's for the most part, will be specifically based on *Paul's* version.

4.2. NLP Records

The primitive data structure in NLP is the *record*. Records are entity-attribute-value elements, largely borrowed from the realm of system simulation [8]. NLP records are based on the belief that objects in the world, *entities*, can be adequately described by their distinguishing properties, *attributes*, and the specific *values* these properties have. In NLP, entities are referred to as records, while attributes and values keep their names.

This approach of entity-attribute-value data structure is very similar to the frame idea [36]. Records are analogous to frames, attributes correspond to slots, and the notion of values is the same for both. Just as a given frame can have more than one slot, an NLP record can have an arbitrary number of attributes. And because the value of a specific attribute for a given record can be another record with its own attributes and values, it is possible to use NLP to implement the information retrieval network speculated about in [36].

There are several ways to implement an entity-attribute-value data structure in LISP. In, *Paul,* property lists were chosen because they seem most natural for this application. Therefore, each record can be thought of as a property list where the attributes are properties and the values are of course the corresponding property values. Using property lists for records necessitates each record to have a unique name, either supplied by the user, in which case the record is called a *named record,* or generated by the system when it creates the record. This requirement would not be found in a version that might use another implementation of records such as association lists. However, it was found that having a name for every record was more of a benefit than a burden. In the act of debugging, both of the code for the NLP system and of subsequent NLP programs, it has often been necessary to examine the contents of specific records, and these records always having readily obtainable names have made them immediately accessible.

4.3. Augmented Phrase Structure Rules

As mentioned earlier, NLP uses production rules [46] to manipulate and generate text. In many ways, this is a logical choice of methods. Many linguistic theories of grammars, including transformational grammars pioneered by Chomsky [4], employ phrase structure rules, which are generally replacement rules. If a specific set of elements is encountered under the proper circumstances, the set is replaced with another. Production rules follow exactly the same format. If a certain situation exists, then a specific action is to be performed. Therefore, production rules are a natural choice for implementing natural language.

To reflect this natural correspondence between production rules and linguistic grammars, the syntax of NLP is very similar to the syntax of phrase structure rules [4]. A typical phrase structure rule might be

```
    SENTENCE::=NOUNPHRASE VERBPHRASE
```

which says that when a SENTENCE is encountered, replace it with a NOUNPHRASE followed by a VERBPHRASE. The equivalent NLP rule might be

```
    SENTENCE  -->  NOUNPHRASE VERBPHRASE;
```

This rule can be read as: "If a record associated with the segment type[6] SENTENCE is encountered, replace it with a record of the segment type NOUNPHRASE followed by a record of the segment type VERBPHRASE."

The syntax for NLP rules as explained up to this point is far too restricted to be useful. An example will clearly demonstrate this, and help provide the motivation for the chosen solution in NLP. If we were to write a set of rules for generating "The boy flies the kite." we might try the following. Ignoring for the now the problem of inserting the actual words into the structure, using the program fragment of Figure 4-1, we could easily construct the following tree.

[6]*Segment type* in NLP corresponds to *symbol* in phrase structure rules, both terminal and nonterminal.

```
SENT  -->  NOUNPHRASE VERBPHRASE . ;
NOUNPHRASE  -->  DETR NOUN;
VERBPHRASE  -->  VERB NOUNPHRASE;
```

Figure 4-1: Fragment of an NLP Program

```
                         SENT

              NOUNPHRASE            VERBPHRASE

          DETR         NOUN         VERB     NOUNPHRASE

                                             DETR      NOUN

          THE           BOY         FLIES     THE      KITE
```

Figure 4-2: Generated Phrase Structure Tree

Now what happens if, instead of the example sentence, we wanted to say "John flies the kite."? Our rules insist that every nounphrase consists of a determiner (*DETR* in the rules) and a noun. Therefore, if we tried to generate this sentence, we would get "The John flies the kite.", which is not at all what we want. A possible solution would be to allow more than one rule for each nonterminal, and adding the following rule to our set.

```
NOUNPHRASE  -->  NOUN;
```

Now we could generate "John flies the kite." However, "The John flies the kite." is still possible from our rules, and now such sentences as "Boy flies kite." can be generated. Adding more rules by themselves is not the answer.

One might think that the problem comes from using the nonterminal symbol NOUNPHRASE for noun phrases both with and without determiners, and that distinct nonterminal symbols for the two distinct phenomena would provide a solution. In addition to losing significant generalities by using such a scheme, the logical conclusion of this is to have a separate nonterminal for every possible terminal string, an impossible feat since there are an infinite number of possible sentences. If one restricts the number of sentences to those anticipated as needed, one is actually providing an elaborate system of canned messages, which we have already dismissed in Chapter One as impractical and linguistically uninteresting.

Somehow, we must be able to choose from among the various rules for each nonterminal symbol. NLP does this through the use of *augmented* phrase structure rules [13, 45]. A list of *condition specifications* is allowed after the segment type to the left of the arrow, the one being replaced. The syntax for condition specifications is quite rich, allowing the user to test for the presence or absence of specific attributes, whether or not attributes have specific values, whether the attribute values of given records are the same or different from the attribute values of other records, whether or not records can inherit specific properties, and just about any other condition the user might care to test.

Furthermore, augmented phrase structure rules allow the user to specify how the records that will replace the original will be created by using a list of *creation specifications*. Again, the syntax is rich, and the options are myriad. All this results in the fact that NLP rules are *not* merely rewrite rules, changing the labeling of a record from one segment type to another, but that they *create new records* for the new segment types. A more complete example (which we will begin to understand as each syntactic option is explained) might be

```
SENT(PASSIVE)  -->  NOUNPH(%GOAL(SENT))
                    VERBPH(%SENT,NUMB:=NUMB(GOAL),-GOAL)
                    # B Y NOUNPH(%AGENT(SENT)) . ;
```

Figure 4-3: Example of NLP Rule

4.4. Condition Specifications

The condition specifications form a series of tests which the current record must satisfy before the rule can be triggered. These tests are mostly variations on determining whether specific attributes have desired values. The simplest test is whether an attribute has any value at all, which is specified by merely naming the attribute to be tested. The example of Figure 4-3 demonstrates this. If the SENT record has any non-false value for the attribute PASSIVE, the rule will be triggered. To understand why the value has to be "non-false" instead of simply "true," we must recall how Boolean logic works in LISP. Rather than testing for true or false, LISP conditional statements test for *NIL* (false) or non-NIL. Non-NIL values are *not* restricted to T (true), but can have *any* value other than NIL. While the PASSIVE attribute *could* have a value of T, any non-NIL value is sufficient to trigger the rule. *RECORD* is a variable whose value will be the name of the current record during execution time. It is through this variable that the system accesses the current record to see if it has the appropriate attribute (by seeing if it has a non-NIL value for the appropriate property).

The NLP syntax also allows the user to specify tests on records other than the one currently on the stack. Recall that the value for an attribute of a given record might itself be a record with its own attributes and values. The user is able to access this "nested" record and its attributes for tests, too. By following the test attribute with a parenthesized *pointer value* (or PV) to another record, the user informs the system that an indirect test is to be performed. For example

```
    EXAMPLE(ALPHA(BETA))
```

might be read as: "If the current record has a segment type of EXAMPLE and the ALPHA value of the record which is the BETA value of the current record is non-NIL, then trigger the rule." In other words, the BETA value of the current record will itself be a record, say RECORD2. If the ALPHA value of RECORD2, *not* the ALPHA value of the current record, is non-NIL, then the rule is to be triggered. This is implemented in LISP through nested GET statements.

```
    (GET (GET *RECORD* 'BETA) 'ALPHA)
```

The syntax is not limited to a single nesting of attributes. The user can specify as many levels as she wants.

```
    EXAMPLE(ALPHA(BETA(...(OMEGA)...)))
```

becomes

```
    (GET (GET (...(GET *RECORD* 'OMEGA)...) 'BETA) 'ALPHA)
```

Notice that the nested hierarchy has originated with the current record *RECORD* in all of the above examples. This is not required by NLP, but is the default origin. The user can specify a named record by following the attribute with the parenthesized name enclosed in single quotes. Therefore the test

```
    EXAMPLE(ALPHA('LETTERS'))
```

is read as: "If the current record has a segment type EXAMPLE, and the named record LETTERS has a non-NIL, ALPHA value, trigger the rule." Nested attribute references are also allowed with specific named records, such as

```
    EXAMPLE(ALPHA(BETA(...(OMEGA('LETTERS'))...)))
```

Collectively, these attribute calls are known as *attribute references*.

In addition to determining whether an attribute reference has a non-NIL value, the user can test for a specific value. This is written by placing an equal sign between the attribute reference and the specific value, which is enclosed in single quotes. It doesn't matter which precedes and which follows the equal sign. Therefore

```
    EXAMPLE(ALPHA='BETA')
```

and

```
    EXAMPLE('BETA'=ALPHA)
```

are logically equivalent. Any of the forms of attribute referencing discussed above are allowed in these equality tests. Additionally, the user can test if the values of two attribute references are equal.

```
EXAMPLE(GAMMA=DELTA)
```

Again, any legal attribute reference may be used here.

There is only one standard attribute in NLP, the SUP attribute. Heidorn conceptually arranged his records into SUPersets, and each record's SUP attribute specifies the superset that record belongs to. A superset is a more general class of entities to which a record belongs, and corresponds to the idea of superordinates. For instance, a record representing "BRIDGET" could have a SUP attribute pointing to a record for "FEMALE." The FEMALE record might have "PERSON" for its SUP attribute, and so on. Such a series of SUP values is known as a *SUP chain.* This specific chain could be interpreted as saying BRIDGET has FEMALE as a superordinate, FEMALE has a superordinate of PERSON, and so forth. The notion of supersets and a specific attribute SUP to represent them is similar to the "AKO" or "a-kind-of" slot that has been suggested for implementing frames [47].

Since the SUP attribute is so prevalent in record definitions, NLP has several conventions for facilitating their use. In attribute tests, rather than explicitly specifying that a value is to be compared to the current record's SUP attribute by using the syntax described above, the user can simply give the value within single quotes. NLP will assume the value is meant for the SUP attribute by default. Therefore

```
EXAMPLE('ALPHA')
```

is exactly equivalent to

```
EXAMPLE(SUP='ALPHA')
```

In addition to nested attribute referencing, NLP allows *indirect* attribute referencing. Frequently, the user may want to use the value of an attribute reference as part of another attribute reference. For example, assume the value of the ALPHA attribute of the current record is either BETA or GAMMA. If the ALPHA value is BETA, the user wants to test the BETA value of the current record. On the other hand, if the ALPHA value is GAMMA, the user wants to test the current record's GAMMA value. In LISP this would be

```
(GET *RECORD* (GET *RECORD* 'ALPHA))
```

The difference between this and the nesting discussed above is that before we were nesting the GET statements along the first argument, the atom, while now we're nesting the GET statements along the second argument, the property.

The user specifies this second kind of nesting by using the commercial at sign, "@". The @ symbol tells the system that the following attribute reference, enclosed in brackets, is an indirect reference. Returning to our previous example, the NLP statement that would represent this test is

```
EXAMPLE(@[ALPHA])
```

Any legal attribute reference can be included between the brackets (including another indirect reference with an @ symbol), and an indirect reference with an @ symbol may be used wherever an attribute is expected. The @ symbol completes the syntax for attribute references.

Another potentially confusing but extremely important test is that of chaining. Recall that records are conceptually arranged into supersets, with each record's SUP attribute specifying the superset that record belongs to, and that a series of SUP values forms a SUP chain. Frequently, it is necessary to determine if the current record belongs to a specific superset, that is, if the name of the superset is anywhere on the current record's SUP chain.

A concrete example should clarify this. Returning to our record for BRIDGET, we remember that its SUP is FEMALE, and the SUP for FEMALE is PERSON. Assume that PERSON has a SUP of HUMAN, that the SUP for HUMAN is MAMMAL, and that the MAMMAL record's SUP is ANIMAL. In other words, we are saying that BRIDGET is a FEMALE, all FEMALES are PERSONS, all PERSONS are HUMANS, all HUMANS are MAMMALS, and all MAMMALS are ANIMALS. We are now ready to test along this SUP chain.

The symbol for tests along chains is the dollar sign, "$". A test to see if the current record is a member of the MAMMAL superset of the MAMMAL superset might be

```
    EXAMPLE($='MAMMAL')
```

When this test is executed, the SUP of the current record is compared to MAMMAL. If they are equal, the test returns T. Otherwise, we move up one on the SUP chain, and test *that* record's SUP with MAMMAL. If that test fails, we again move up one on the chain. This process continues until either a record is found on the chain whose SUP is equal to MAMMAL, in which case the test succeeds, or until the end of the SUP chain is reached by encountering a record with no SUP value, in which case the test fails. Conceptually in LISP what we want is

```
    (OR (EQUAL (GET (*RECORD* 'SUP) 'MAMMAL)
        (EQUAL (GET (GET *RECORD* 'SUP) 'SUP) 'MAMMAL)
        (EQUAL (GET (GET (GET *RECORD* 'SUP) 'SUP) 'SUP) 'MAMMAL)
        ...)
```

As with any equality test, the ordering of the chaining reference and the value being tested for is *not* crucial. Therefore,

```
    EXAMPLE($='MAMMAL')
```

and

```
    EXAMPLE('MAMMAL'=$)
```

are equivalent. Furthermore, the value being tested for can take the form of any of the attribute references we have already seen. Thus

```
EXAMPLE($=ALPHA(BETA('LETTER')))
```

is completely legal. Additionally, we can specify the search to start elsewhere than the current record. This is done by placing the desired attribute reference immediately before the $. An example might be

```
EXAMPLE(ALPHA(BETA)$=ONE(TWO))
```

In addition to seeing whether a record is a member of a superset, it is often necessary to test whether the record or any member of its superset has a specific value for an attribute other than the SUP attribute. This brings in the notion of *inheritance* [47]. Returning to our BRIDGET SUP chain example, we know that mammals exhibit certain traits that are not generally found in every animal. For instance, mammals are warm blooded. Since Bridget is a mammal, she is also warm blooded. If we wanted to include this fact in our system, we could add a BLOOD attribute to the BRIDGET record with a value of WARM. However, we would then have to explicitly include this attribute and same value for every record that is a member of the MAMMAL superset. It would be much more general to give the MAMMAL record the BLOOD attribute and the WARM value. Then every member of the MAMMAL superset could *inherit* this attribute and value. That is, every member of the superset is known to have the attributes and values of the superset, including the BLOOD attribute with the WARM value, unless we are told explicitly otherwise.

In order to test if the current record can inherit the value for the WARM attribute, the following NLP syntax is used.

```
EXAMPLE($['BLOOD']='WARM')
```

The brackets inform the system that an argument is being given to the chaining function called for by the $. It is important to think of this as a function with an argument. Notice that the BLOOD attribute in the example is in single quotes. This is necessary because we want to use the literal BLOOD as the argument to the chaining function. If BLOOD were not in quotes, the value of the current record's BLOOD attribute would be given as an argument to the chaining function. As always, any attribute reference can be used within the brackets.

Actually, the chaining function always has this argument. When it isn't supplied explicitly by the user, as we saw when the $ was first introduced, the argument defaults to SUP. Therefore

```
EXAMPLE($='MAMMAL')
```

and

```
EXAMPLE($['SUP']='MAMMAL')
```

are identical. Again, ordering around the equal sign is unimportant, and any attribute reference can serve as the value being tested as well as the starting point for the chaining test.

In addition to having chains along the SUP attribute, there's no reason why the records can't have chains along other attributes, and there's no reason why the chaining function can't use these other chains. By supplying the function with a second argument, the user can specify which attribute chain she wants to exploit. An example might be

```
    EXAMPLE($['ALPHA','BETA']='GAMMA')
```

This says to chain along the BETA attribute, looking for a record whose ALPHA value is equal to GAMMA. Notice that the second argument is also in quotes for the same reasons that the first is (as explained above), and that the arguments are separated by a comma.

As with the first argument, the second argument is required by the chaining function, and when it is not explicitly supplied, the argument defaults to SUP. Therefore,

```
    EXAMPLE($['BLOOD']='WARM')
```

and

```
    EXAMPLE($['BLOOD','SUP']='WARM')
```

are equivalent, as are

```
    EXAMPLE($='WARM')
```

and

```
    EXAMPLE($['SUP','SUP']='WARM')
```

An important restriction on the second argument is that it can *not* be specified if the first argument is not. If the user wants to specify the second argument, she *must* supply the first, even if it is to be SUP.

Finally, the user can specify the record the chain is to start with by giving the appropriate attribute reference or literal before the dollar sign. As usual, if none is supplied, the system defaults to the current record being tested.

This completes the syntax for chaining references. The first argument, whose value is the attribute being tested for along the chain, is known as the *test attribute*, while the second, whose value is the attribute being chained along, is called the *chain attribute*.

Just as we could use attribute references without equal signs to test if they had *any* non-NIL value, we can use chaining references without equal signs to test if they return any non-NIL values. In this case, the first non-NIL value for the test attribute found along the chain specified by the chain attribute is returned. If none is found, NIL is returned and the test fails. If the BRIDGET record were the current record, execution of the test

```
    EXAMPLE($['BLOOD'])
```

would return WARM, assuming none of the records between BRIDGET and MAMMAL had a non-NIL BLOOD value. The same defaults and restrictions described above for chaining references apply for this use of them. A good test to see whether chaining references and their defaults are understood would be to describe what is specified in the following test.[7]

```
EXAMPLE($)
```

Finally, two chaining references may be used in the same equality test. An example could be

```
EXAMPLE($['ALPHA','BETA']=$['ONE','TWO'])
```

This says that if the ALPHA value inherited along the BETA chain of the current record is equal to the ONE value inherited along the TWO chain, the test succeeds.

So far we have only seen tests consisting of a single condition specification. NLP allows the user to combine an arbitrary number of condition specifications into a single test. One way is to separate condition specifications by commas. This has the effect of inserting logical ANDs between each individual test.

```
EXAMPLE(ALPHA,BETA)
```

becomes in LISP

```
(AND (GET *RECORD* 'ALPHA)
     (GET *RECORD* 'BETA))
```

Any of the types of condition specifications discussed above are allowed, as well as any number of condition specifications in a single test.

By placing a vertical bar "|" between two condition specifications, the user states that *either* the first test OR the second is sufficient to trigger the test.

```
EXAMPLE(ALPHA|BETA)
```

As with AND, any number and type of condition specifications can be combined with OR.

Logical ANDs and ORs may be combined in the same test.

```
EXAMPLE(ALPHA,BETA|GAMMA)
```

becomes

```
(AND (GET *RECORD* 'ALPHA)
     (OR (GET *RECORD* 'BETA)
         (GET *RECORD* 'GAMMA)))
```

Notice that the vertical bar OR has precedence over the comma AND. This is true throughout condition specifications in this NLP system. Heidorn's version did not explicitly address the question of precedence, and

[7]ANSWER: With the defaults, this test becomes

```
EXAMPLE($['SUP','SUP'])
```

and says to test the current record for a SUP value along the SUP chain. In other words, if the current record has any non-NIL SUP value, the test succeeds. Otherwise go to the record specified by the current record's SUP value and repeat the test, continuing until either a record is found whose SUP value (as the test attribute) is non-NIL and the test succeeds, or until its SUP value (as the chain attribute) is NIL and the test fails. Obviously, either the current record has a non-NIL SUP value for the test attribute, in which case the test immediately succeeds *without* chaining, or it has a NIL SUP value for the chain attribute, in which case the test immediately fails because it *can't* chain. In either case, no chaining is performed. This test is therefore identical to

```
EXAMPLE(SUP)
```

his resulting Boolean operators have an ad hoc precedence. When the current version of NLP was developed, it was felt that an explicit precedence would help create uniform rules.

While an explicit precedence exists in this system, the user can override it through the standard use of parentheses. Therefore,

```
     EXAMPLE((ALPHA,BETA)|GAMMA)
```

becomes

```
     (OR (AND (GET *RECORD* 'ALPHA)
              (GET *RECORD* 'BETA))
         (GET *RECORD* 'GAMMA))
```

giving the comma AND precedence over the vertical bar **OR.** Superfluous parentheses are ignored, provided they are correctly balanced.

Completing the Boolean entourage is the logical NOT. The current system allows two symbols, the caret, "↑", and the tilde, "~", to be used for NOT. Both the caret and the tilde perform the exact same function. (In fact, the system converts all tildes to carets before processing rules.) The tilde was included to accommodate users who were accustomed to the tilde as the symbol for logical NOT. A NOT symbol before any condition specification states that the test is to succeed if and only if the condition specification fails.

```
     EXAMPLE(↑ALPHA)
     EXAMPLE(↑(ALPHA=BETA))
     EXAMPLE(↑$['ONE','TWO'])
```

NOT has precedence over *AND* and OR, but parentheses can again override this. So while

```
     EXAMPLE(↑ALPHA,BETA)
```

becomes

```
     (AND (NOT (GET *RECORD* 'ALPHA))
          (GET *RECORD* 'BETA))
```

the following test

```
     EXAMPLE(↑(ALPHA,BETA))
```

becomes

```
     (NOT (AND (GET *RECORD* 'ALPHA)
               (GET *RECORD* 'BETA)))
```

The Boolean operators complete the syntax for condition specifications as explained in the original report.

NLP has been extended since that report, however. One of the extensions included in *Paul's* version of NLP is the exclamation point, "!". When the system encounters an !, the element immediately following it is treated as a LISP s-expression, *not* an NLP element. Therefore

```
     EXAMPLE(ALPHA,!(NUMBERP BETA))
```

becomes

```
     (AND (GET *RECORD* 'ALPHA)
          (NUMBERP BETA))
```

The s-expression is inserted directly into the LISP code just as it appeared in the rule.

Another addition that has been added is to allow *explicit function calls* in NLP rules. The system recognizes function calls by the parameter list enclosed in angle brackets "<>" directly following the function name. The distinction between this kind of function call and the insertion of a LISP function directly into the code through the use of ! is that the parameters in the angle bracket list are *NLP attributes and are resolved in the normal way.* An example might help to make this clear.

```
    EXAMPLE(!(NUMBERP ALPHA))
```

as we know, simply becomes

```
    (NUMBERP ALPHA)
```

On the other hand, in our new function call,

```
    EXAMPLE(NUMBERP<ALPHA>)
```

ALPHA is treated as an attribute, and the result is

```
    (NUMBERP (GET *RECORD* 'ALPHA))
```

This allows the user to use function calls with attribute values as parameters *without* requiring her to know these values ahead of time.

The user is also allowed to have a segment type without condition specifications. In this case, *any* record of this segment type would trigger the rule. The syntax for the condition part of NLP rules is now complete.

4.5. Creation Specifications

In addition to specifying the conditions under which a rule is to be triggered, augmented phrase structure rules allow the user to designate the specifications for creating a new record. The first element of this part of an NLP rule is a segment type. This is the segment type that will be associated with the new record when it is placed on the control stack, and will be used when it's that record's turn to trigger rules.

Following the segment type is an optional list of *creation specifications* which spell out in detail how the new record is to be created. The syntax for many of the creation specifications is similar to that for condition specifications, but the meaning is slightly different. The simplest creation specification is once again the name of an attribute.

```
    EXAMPLE(ALPHA)
```

However, rather than testing to see if the current record has a non-NIL value for the ALPHA attribute, here we want to *assign* a non-NIL value to the *new record's* ALPHA attribute. In other words, instead of retrieving a property value with a GET statement, we want to assign a property value with a PUTPROP statement. Since the user hasn't specified the value to be assigned, only that it be non-NIL, the system uses the simplest non-NIL value available, namely T. As with condition specifications, pointer values may be used.

```
    EXAMPLE(ALPHA(BETA))
```

is legal and is read as "Create a new record of segment type EXAMPLE whose ALPHA value of the record which is this new record's BETA value is T." In other words, the BETA value of the new record is obtained, and *that* record's ALPHA value is set to T. The LISP code to do this is

```
    (PUTPROP (GET *RECORD* 'BETA) T 'ALPHA)
```

Of course, if the new record does not yet have a BETA value, the nested GET will return NIL and the command will put the value T and the property ALPHA on the property list of NIL, which is probably not what the user intended. Notice that whether or not the new record has a BETA value, the property list of the new record is *not* affected in any way. This means the rule would have no direct effect on the new record.

Since this kind of specification is usable on all types of attribute references, it can be used on those for specifically named record, and with indirect referencing.

```
    EXAMPLE(ALPHA('LETTERS'))

    EXAMPLE(@[ALPHA])
```

There is one convention in creation specification attributes that is not found in those of condition specifications. In condition specifications, the default record was the current one being tested. In creation specifications, the default is the record being created. In order to use attributes of the record that caused the rule to trigger, NLP has the convention of using the segment type of that record, ***the condition segment type***, for the name of this triggering record. As an example, if we had a rule whose condition segment type were SENT, then

```
    EXAMPLE(@[ALPHA(SENT)])
```

would become

```
    (PUTPROP *RECORD* T (GET *OLD-RECORD* 'ALPHA))
```

At the time of execution, the LISP variable *RECORD* will still contain the name of the record being created, and *OLD-RECORD* will contain the name of the record that triggered this rule.

Just as NLP allows the testing of attribute references for specific values, the system allows the assignment of specific values in creation specifications. The operator for assignment, ":=," immediately follows the attribute reference that is to receive a value, and the actual value to be assigned next appears. For instance,

```
    EXAMPLE(ALPHA:='BETA')
```

says to create a new record of segment type EXAMPLE with an ALPHA attribute whose value is BETA. In addition to literals, attribute references may be used, since they eventually return values.

```
    EXAMPLE(ALPHA:=BETA)
```

Any legal attribute references, including arbitrary nesting, indirect references through the use of the @ symbol, and using explicitly named records are allowed, and these attribute references use the same syntax that condition specifications use.

One significant difference, though, is that order around the assignment operator is crucial. Unlike equality tests, where, as in all tests, no record is actually being altered, assignment clearly changes the value of an attribute for some record. The left part of an assignment designates where a new value is to be stored, and the right part states what that value is. The two parts serve very different purposes, and the assignment operator, unlike the equality operator, is therefore *not* symmetric. Furthermore, the left part of an assignment must be able to receive a value. In other words, it must be an attribute reference, not a literal in single quotes, a chaining reference, or an explicit function call.

As there was a simplified syntax for testing a record's SUP value, there is a simplified syntax for assigning a value to a record's SUP attribute. The syntax is the same, the literal value simply appearing within single quotes.

```
    EXAMPLE('ALPHA')
```
is equivalent to
```
    EXAMPLE(SUP:='ALPHA')
```

Chaining can also occur in creation specifications. The same symbol, the dollar sign, is used for chaining, and the same syntax, defaults, and restrictions for chaining references in condition specifications apply to their use in creation specifications. Additionally, since chaining references can only obtain values, they can *not* receive values; they can only appear on the right side of assignments (unless they are being used as an indirect reference within an @).

The exclamation point extension discussed above may be used in creation specifications as well as in condition specifications. The LISP s-expression immediately following the ! is read in as such and is placed as is directly into the LISP code being generated. The NLP system does not attempt to convert the s-expression into LISP (it already *is* in LISP), nor is the s-expression evaluated at this time.

There is an additional creation specification operator that has no corresponding condition operator, the per cent sign, "%". Frequently, the user will want to give a new record *all* the attributes and values of some other record. Listing each attribute assignment individually is too cumbersome, and there is no reason to assume that the user will know at the time the rule is being written every attribute the record being copied will have at the time of execution. The % solves this problem. The % followed by any legal attribute reference tells the system to copy into the new record *all* the attributes and corresponding values of the record pointed to by the attribute reference. For instance,

```
    EXAMPLE(%ALPHA('LETTERS'))
```
copies the entire record found at
```
    (GET 'LETTERS 'ALPHA)
```
into the new record. That is, each property found on the plist of the item returned by (GET 'LETTERS

'ALPHA) is put onto the plist of *RECORD* with the same property value.

There is also an automatic use of the copying function. If the segment type of the record being newly created is the same as the segment type of the record that triggered this rule, the triggering record is automatically copied into the newly created record as the first action of the creation specification. The only time this *doesn't* occur is when the creation specification of the new record has an explicit command to copy some record (signified by the use of the % operator).

More than one creation specification may be included in the same list by separating them with commas. The actions designated by the creation specifications are performed sequentially from left to right. As an example,

```
    EXAMPLE('ALPHA',BETA:=BETA('LETTERS'))
```

says the following: "Create a new record of segment type EXAMPLE, assign the value ALPHA to the SUP attribute of this new record, and assign to its BETA attribute the BETA value of the named record LETTERS."

After copying an existing record into the new one, the user has no problem adding or reassigning attributes to the new record.

```
    EXAMPLE(%'LETTERS',ALPHA:='ONE')
```

copies the attributes of the record LETTERS into the new record, then changes the new record's ALPHA value to ONE. However, the user will frequently want to eliminate or "turn off" some attribute after copying a record. She can do this by using the minus sign or hyphen, "-". A hyphen followed by an attribute reference tells NLP to remove that attribute reference, giving it a NIL value. Therefore

```
    EXAMPLE(-ALPHA)
```

becomes

```
    (REMPROP *RECORD* 'ALPHA)
```

The hyphen can be used with any legal attribute reference.

```
    EXAMPLE(-ALPHA(BETA))

    EXAMPLE(-ALPHA('LETTERS'))
```

Earlier it was said that the list of creation specifications following the segment type is optional. If no list of creation specifications is given, the segment type is pushed onto the control stack *without creating a new record to be associated with this segment type.* When such a segment type is encountered, the system treats it as a terminal symbol, and the segment type is placed directly into the output stream.

4.6. The Complete NLP Rule

Now that we know the syntax for the individual parts of an NLP rule, let's see what the format is for putting these parts together. An NLP rule is made up of the condition part (consisting of a segment type followed by an optional list of condition specifications), followed by an arrow and then one or more creation parts (each consisting of a segment type followed by an optional list of creation specifications). A new record is created for *each* creation part that calls for it (by having a list of creation specifications), and these records are pushed onto the control stack with the first one being created on top. The arrow consists of a greater-than symbol ">" preceded by at least one hyphen "-". This arrow is not strictly necessary, since in a rule the only thing allowed on the condition side after the segment type is at most one list of condition specifications enclosed in parentheses, and the first element of the creation side of a rule must be another segment type, there can be no ambiguity as to where the condition part stops and the creation part starts. However, the arrow improves readability of rules, especially complicated ones in which both condition and creation specifications take up several lines, and the arrow helps make the analogy between NLP rules and phrase structure rules more apparent.

The exact number of hyphens in the arrow is not important, as long as there is at least one. This allows the user to line up her rules as she wants them, in effect permitting so-called "pretty printing." Additionally, spaces, line feeds, and returns are ignored by the input system, further enhancing the user's capability to pretty print. In fact, spaces, line feeds, and returns are ignored throughout the NLP system. Consequently, the user must tell NLP when a rule (or *any* input segment) is finished. She does this by ending each input segment with a semi-colon. Ending a rule with an empty bracket list "[]" tells NLP that not only is the rule finished, but this is in fact the last rule to be processed.

NLP also allows the user to put comments in her rules. These comments are delimited by braces "{}" and may appear anywhere within or between rules. Everything within the braces will be ignored by the system.

Let's return to our first example of a complete NLP rule.

```
SENT(PASSIVE)  -->  NOUNPH(%GOAL(SENT))
                    VERBPH(%SENT,NUMB:=NUMB(GOAL),-GOAL)
                    # B Y NOUNPH(%AGENT(SENT)) . ;
```

Figure 4-4: Example of NLP Rule

This says: "If the current record (call it TRIGGER) has a segment type of SENT and a non-NIL PASSIVE value, do the following. Create a new record (call it RECORD1) which is a copy of the record found in the GOAL attribute of TRIGGER (recall that in the creation part the condition segment type (SENT in this case) is the convention for referring to the record that triggered the rule), and associate this record with a segment type of NOUNPH. Create a second record (call it RECORD2) by copying TRIGGER, assigning to the NUMB attribute of RECORD2 the NUMB value of the GOAL of RECORD2, next removing the GOAL attribute from RECORD2 (note that the order in which these operations are performed is critical), and associate RECORD2 with the segment type VERBPH. Insert each of the segment types, #[8], B, and Y onto the stack *without* creating records for them. Create a last record (call it RECORD3) by copying the AGENT value of TRIGGER, and give RECORD3 a segment type of NOUNPH. Finally, insert a period onto the stack without a record." If the control stack consisted of

```
((SENT TRIGGER))
```

before executing this rule , it would be

```
((NOUNPH RECORD1) (VERBPH RECORD2)
 (#) (B) (Y) (NOUNPH RECORD3) (.))
```

after execution.

4.7. Named Records

In addition to writing rules to test, create, and manipulate records, NLP allows the user to explicitly define named records for her program to use. Since this action is in reality the creation of records, it is virtually the same as the creation part of an NLP rule. Therefore the syntax for creating records is nearly identical to that of the creation part of NLP rules. The main difference is that the record definition starts with the name for the record rather than an associated segment type. Therefore if the following were a record definition

```
EXAMPLE('ALPHA',BETA:='GAMMA');
```

the named record EXAMPLE would be created with a SUP value of ALPHA and a BETA value of GAMMA. The specifications for a record definition are identical to creation specifications. As with rules, record definitions must end in either a semi-colon or an empty bracket list.

4.8. Cover Attributes

Finally, the user is allowed to define what are called *cover attributes*. Frequently a set of attributes can be logically grouped together. For instance, the attributes MALE, FEMALE, and NEUTER all refer to GENDER. While specifications could be explicitly written to deal with each of these, it would be more convenient when the same action is to be performed on each of these attributes if we could write one specification to perform all these operations. Additionally, it would make clear the fact that these attributes are associated in some way.

[8] As we shall see later, NLP uses this symbol in the output stream to represent a space.

By defining cover attributes, the user can associate attributes this way. The user defines cover attributes by giving the name of the cover attribute followed by a list of the attributes to be grouped together under this "cover." Again, each definition ends with a semi-colon except the last, which ends with an empty bracket list. Any attribute name is allowed in this list, including another cover attribute.

```
GENDER (MALE FEMALE NEUTER);
PRONOUNS (GENDER NUMBER CASE);
COVER1 (ALPHA BETA GAMMA);
COVER2 (ONE TWO THREE)[]
```

When NLP encounters a cover attribute in either a rule or a record definition, the specification is replaced by specifications containing each of the attributes being covered. For instance, using the COVER1 and COVER2 examples from above,

```
EXAMPLE(COVER1)
```

as a creation specification becomes

```
EXAMPLE(ALPHA,BETA,GAMMA)
```

That is, in the newly created record, the attributes ALPHA, BETA, and GAMMA will each have a value of T.

```
EXAMPLE(COVER2:='NUMBER')
```

would become

```
EXAMPLE(ONE:='NUMBER',TWO:='NUMBER',THREE:='NUMBER')
```

In condition specifications, cover attributes are handled slightly differently. A creation specification test involving a cover attribute will succeed if the specified test succeeds for *any one* member of the cover attribute. That is,

```
EXAMPLE(COVER1)
```

will succeed if ALPHA is non-NIL, *or* if BETA is non-NIL, *or* if GAMMA is non-NIL. Instead of replacing COVER1 with its member attributes separated by commas, which signifies ANDS, COVER1 is replaced by these attributes separated by vertical bar **ORS.**

```
EXAMPLE(ALPHA|BETA|GAMMA)
```

If the cover attributes appear on both sides of an equality test or an assignment, first one cover attribute, then the other is expanded, resulting in every possible pairing of the attributes.

```
EXAMPLE(COVER1=COVER2)
```

would be expanded into nine tests, and

```
EXAMPLE(COVER1:=COVER2)
```

would become nine separate assignments. An exception to this is when the same cover attribute is used on both sides. Then, rather than expanding to every possible combination, only the pairings of the same attribute on both sides are used.

```
EXAMPLE(COVER1:=COVER1('LETTERS'))
```

becomes

```
EXAMPLE(ALPHA:=ALPHA('LETTERS'),
        BETA:=BETA('LETTERS'),
        GAMMA:=GAMMA('LETTERS'))
```

Note that in the above example the second use of the cover attribute referred to a named record. Cover attributes can be used wherever a regular attribute name is allowed.

Because cover attributes affect the meaning of rules and record definitions that they appear in, any cover attributes to be used must be defined before rules or records.

4.9. Record Definitions

NLP allows the user to create records very easily . Each record definition consists merely of the name of the record followed by a list of attributes and their values in parentheses. Since we are creating records, which is exactly the same function performed by creation specifications of rules, we would want record definitions to have the same syntax and meaning as creation specifications. This is precisely the case. All the syntax for creation specifications, including assignment of attributes, default assignment for SUP, use of %, @, $, !, and cover attributes, is allowed in record definitions with exactly the same meaning.

As an example, suppose we wanted an record to represent our friend Bridget. We might want to note that she is female, that she has blond hair, that her age is four, and that she is alive. The following record definition would accomplish this:

```
BRIDGET ('FEMALE',HAIRCOLOR:='BLOND',AGE:='4',ALIVE);
```

This definition would create a record BRIDGET with a SUP attribute whose value is FEMALE, a HAIRCOLOR with a value of BLOND, an AGE attribute with a value of 4, and an ALIVE attribute whose value is **T**.

A typical program first defines the cover attributes to be used. Next the actual rules to be executed are given. Then any named records the user wants are defined. The control stack is initialized with record(s) and their associated segment types. Finally, the user invokes her NLP program to *encode* or generate text. The command BYE then leaves the NLP system and returns the user to the host environment. Appendix IV contains a BNF for NLP, and Appendix V contains the complete NLP program for *Paul*.

4.10. The Generation Algorithm

Now that we know how to write NLP rules, we can see how the system executes these rules, and examine the control mechanisms which determine the order in which the actions of the rules will be performed. The central control mechanism for NLP is a stack of segment types and associated records. As the generation process proceeds, the stack is popped one item at a time. The appropriate action based on the specific item is taken, and the results are either pushed back onto the stack or inserted into the output stream. When the stack is finally empty, the process is finished.

The generation algorithm used in *Paul* is the one found in the original NLP report [13]. This algorithm is repeated here in Figure 4-5.

```
1. Put a segment type name and a record on the stack to begin.

2. Take the top segment type name and associated record (if
   there is one) off the stack, and examine the segment type:

    a. if it is a terminal segment type (known by there not
       being an associated record), put its name into the
       output stream.

    b. if it is one of the special OUTPUT segment types,
       perform the specified output operation.

    c. otherwise, examine each rule that has this segment type
       on the left as the condition segment type until either
       a rule is found for which the conditions specified in
       parentheses are met, or until the list of rules is
       exhausted:

        i. if a rule is found, create segment records
           according to the specifications given in
           parentheses on the right side, and put the segment
           type names, along with their newly created
           associated records, onto the stack.

        ii. otherwise, put into the output stream the value of
            the SUP attribute of the record which was taken
            off the stack.

3. Repeat step 2 until the stack is empty.
```

Figure 4-5: The Generation Algorithm

As is readily apparent, the control algorithm is conceptually simple. The system basically searches through the ordered rules sequentially until it either finds one to use (determined by the condition specification tests applied to the segment record), or the list of rules is exhausted. The associated segment type is used to restrict this search by limiting the rules that are considered to only those that have the correct condition segment type. The first rule of the correct segment type whose conditions are satisfied by the segment record is applied.

It is important to realize that these rules are *not* merely rewrite rules. The significant difference is that this algorithm uses *augmented phrase structure grammar*, which deals with segment records in addition to the segment types, instead of just manipulating and replacing nonterminal and terminal symbols. The difference between augmented phrase structure rules and context free phrase structure rules is similar to the difference between ATNs and RTNs [45]. Just as an ATN has feature registers associated with the nodes of the tree it is building, an augmented phrase structure system has records associated with the nodes of the tree it is building. ATNs have conditions and actions associated with their arcs which can test and modify the contents of feature registers, and augmented phrase structure rules have condition and creation specifications which can modify the contents of the records. These characteristics, which are the chief properties that distinguish ATNs from RTNs, are similarly the main properties that distinguish augmented phrase structure rules from context free phrase structure rules.

In (2b) of the algorithm, special *OUTPUT segment types* are mentioned. Currently, *Paul* has three such segment types. The first one, also appearing in the original NLP report, is the sharp sign "#". This is used in rules to represent a space in the output. Recall that the NLP system ignores spaces, linefeeds, and returns in its input. If the user wants to have a space inserted into the output stream, she cannot simply put a space in the appropriate place in the rule; it will be ignored. Instead, she should put a # there. When this symbol is popped off the control stack by *Paul*, section (2b) of the generation algorithm applies, and a blank is inserted into the output stream. Similarly, if the user wants a linefeed in her text (if she wants to start on a new line, for instance), she again needs a special output segment type, LINE. This will insert a linefeed into the output when encountered. Finally, the special output segment type NULL inserts a NULL string into the output. This is used for rules for "zeroing out" some item, that is, replacing some nonterminal symbol (a segment type and its associated record) with nothing.

4.11. The Generation Paradigm

It is important to distinguish the *generation algorithm* from the *generation paradigm*. The former is the control mechanism behind the selection of the rules, and as such, is an integral part of NLP. But NLP is only a programming language, and as with all programming languages, it can be used in many different ways to perform many different tasks. While an understanding of the language of NLP and the control mechanism that drives it is important, it is not a goal onto itself, but a means for seeing how the language is used. The generation paradigm, on the other hand, is the theoretical base from which *Paul* converts conceptual representations into surface language.

Paul uses augmented phrase structure grammar to construct a syntactic tree in a strict left-right top down fashion. Perhaps the best way to proceed is to present an example, then discuss the various aspects of the paradigm.

```
COVER ATTR;

NUMB   (SING PLUR);
PERS   (PERS1 PERS2 PERS3);
TENSE  (PAST PRESENT FUTURE);
DET    (DEF INDEF DEM POSSESS);
ENDING (ED ING)[]
```

Figure 4-6: Cover Attributes for Example

```
RECORDS;

{vocabulary records}
BUY1  ('ACQUIRE',WORD:='BUY');
JOHN  ('BOY',GENDER:='MALE',PROPER);
KITE  ('TOY');
TOY   ('THING');
THING (GENDER:='NEUTER');
BOY   ('HUMAN');
BUY   (PAST:='BOUGHT');

{sentence records}
A1 ('BUY1',AGNT:='A2',AFF:='A3');
A2 ('JOHN');
A3 (KITE)[]
```

Figure 4-7: NLP Records for Example

Figures 4-6 through 4-8 contain a small NLP program to generate the sentence "John buys a kite." One thing that we notice right away about the rules of Figure 4-8 is that they are recursive. That is, some rules replace segment types with the same segment types. For instance, rule {5},

```
VP(↑NUMB)   -->   VP(SING);
```

replaces a VP (verb phrase) with a VP. The thing that prevents this rule from endlessly looping is the fact that it is *augmented* with both condition and creation specifications. It is not allowed to apply to any record that has a VP segment type. The record must also *not* have a non-NIL value for either of the attributes that are members of the NUMB cover attribute (SING and PLUR). If this is true, then a *new* record is created and associated with the VP segment type. This new record, in addition to having all the attributes of the triggering record (recall that when a segment type on the creation side is the same as the segment type from the condition side, an automatic copy is performed), has the additional attribute of SING with the value T.

```
RULES FOR ENCODING;

{SENT is Sentence}
{1} SENT  -------------------->  NP(%AGNT(SENT))
                                 VP(%SENT,
                                    NUMB:=NUMB(AGNT),
                                    -AGNT) . ;

{NP is Noun Phrase}
{2} NP(↑DET,↑$['PROPER'])  -->  NP(INDEF);
{3} NP(DET,↑DETR)  ---------->  DETR(%NP) NP(DETR);
{4} NP  --------------------->  NOUN(%NP);

{VP is Verb Phrase}
{5} VP(↑NUMB)  -------------->  VP(SING);
{6} VP(↑PERS)  -------------->  VP(PERS3);
{7} VP(↑TENSE)  ------------->  VP(PRESENT);
{8} VP(AFF)  ---------------->  VP(-AFF)
                                NP(%AFF(VP));
    {AFF is for the AFFECTED case role}
{9} VP  --------------------->  VERB(%VP);


{DETR is Determiner}
{10} DETR(INDEF,PLUR)  ------->  NULL;
{11} DETR(INDEF)  ------------>  WORD('A');
{12} DETR(DEF)  -------------->  WORD('THE');

{13} NOUN  ------------------->  NOUNP(%NOUN);

{14} VERB  ------------------->  VERBP(%VERB,SUP:=WORD(SUP));

{NOUNP is Noun Part}
{15} NOUNP(PLUR)  ------------>  WORD(%NOUNP) S;
{16} NOUNP  ------------------>  WORD(%NOUNP);

{VERBP is Verb Part}
{17} VERBP(PLUR|PERS2)  ------>  WORD(%VERBP);
{18} VERBP(PERS1)  ----------->  WORD(%VERBP);
{19} VERBP  ------------------>  WORD(%VERBP) S;

{20} WORD('NULL')  ----------->  NULL;
{21} WORD(E(SUP),↑ENDING)  --->  # OUTPUT(%WORD) E;
{22} WORD  ------------------->  # OUTPUT(%WORD)[]
```

Figure 4-8: NLP Rules for Example

Therefore, Rule {5} is *not* truly recursive. In fact, none of the rules of Figure 4-8 are, nor are they in *Paul* (as can be seen in Appendix V).

That is not to say that augmented phrase structure rules *can't* be recursive. Consider the following rule.

```
SENT  -->  I # AM # VERY # LONG # SENT;
```

If this were the first rule of segment type SENT and a record of segment type SENT ever entered the stack, this rule would be executed without ever halting, resulting in the output, *I AM VERY LONG I AM VERY LONG I AM VERY LONG I AM VERY...* It is only because the rules of *Paul* are carefully defined that such types of recursion, and subtler versions where the recursion loops through several rules, are avoided.[9]

Figure 4-9 is a trace of the stack and rules that would be used in running this program. The stack is intialized with the segment type SENT and the associated record A1. A1 is the deep case structure for the sentence, and contains all the semantic information needed for generation. Figure 4-10 shows the contents (plists) of the records that would be created.

[9]As a consequence of this restriction, certain left-branching sentences cannot be generated by *Paul.* An example of such a sentence is "John's cousin's friend's brother's neighbor knows Marvin Minsky."

RULE	OUTPUT	STACK
		((SENT A1))
1		((NP A2) (VP *1*) (.))
4		((NOUN A2) (VP *1*)) (.))
13		((NOUNP A2) (VP *1*) (.))
16		((WORD A2) (VP *1*) (.))
22		((#) (OUTPUT A2) (VP *1*) (.))
	#	((OUTPUT A2) (VP *1*) (.))
	JOHN	((VP *1*) (.))
5		((VP *2*) (.))
6		((VP *3*) (.))
7		((VP *4*) (.))
8		((VP *5*) (NP A3) (.))
9		((VERB *5*) (NP A3) (.))
14		((VERBP *6*) (NP A3) (.))
19		((WORD *6*) (S) (NP A3) (.))
22		((#) (OUTPUT *6*) (S) (NP A3) (.))
	#	((OUTPUT *6*) (S) (NP A3) (.))
	BUY	((S) (NP A3) (.))
	S	((NP A3) (.))
2		((NP *7*) (.))
3		((DETR *8*) (NP *9*) (.))
11		((WORD *10*) (NP *9*) (.))
22		((#) (OUTPUT *10*) (NP *9*) (.))
	#	((OUTPUT *10*) (NP *9*) (.))
	A	((NP *9*) (.))
4		((NOUN *9*) (.))
13		((NOUNP *9*) (.))
16		((WORD *9*) (.))
22		((#) (OUTPUT *9*) (.))
	#	((OUTPUT *9*) (.))
	KITE	((.))
	.	

Figure 4-9: Trace of Control Stack for Example

```
*1*  (SUP BUY1 AFF A3)
*2*  (SUP BUY1 AFF A3 SING T)
*3*  (SUP BUY1 AFF A3 SING T PERS3 T)
*4*  (SUP BUY1 AFF A3 SING T PERS3 T PRESENT T)
*5*  (SUP BUY1 SING T PERS3 T PRESENT T)
*6*  (SUP BUY SING T PERS3 T PRESENT T)
*7*  (SUP KITE INDEF T)
*8*  (SUP KITE INDEF T)
*9*  (SUP KITE INDEF T DETR T)
*10* (SUP A)
```

Figure 4-10: Created Records for Example Sentence

These rules can be thought of as building a tree from the top down to achieve the proper syntactic surface structure. Figure 4-11 shows the tree for our example sentence.

Because the generated tree has records containing additional information associated with the appropriate nodes, augmented phrase structure rule systems are able to achieve *indelibility* [32, 34]. That is, once a decision has been made and a node is incorporated into the tree, it cannot be taken back. When a decision point is arrived at and not enough information is available at the time to choose the proper path, one of two approaches is commonly taken. The first is to arbitrarily choose one path over the others and proceed. If this decision proves later on to be wrong, the steps taken since then are retraced to that decision point, and another path is selected. This is known as backtracking. The other alternative is to explore *all* the paths simultaneously, abandoning only those which prove to be dead ends, until finally one is discovered as the true way. An indelible system is one that avoids both backtracking and parallel expansion by insuring that all the necessary information is available at the time the decision has to be made.

Computationally, an indelible system is to be preferred. In memory considerations, an indelible system is obviously more efficient than one that runs choices in parallel because competing paths do not have to be maintained until the correct one is found. Indelible systems are also superior to backtracking systems in this respect. Backtracking systems typically need to remember decision points and the options available at each one. Furthermore, they need to remember the *state* they were in at each decision point, and must undo all actions after the decision point when backtracking. The solutions to these problems require both memory and computational time, while indelible systems avoid the problems altogether.

There are two chief reasons why *Paul* is able to maintain indelibility. The first is that its augmented phrase structure rules generate trees augmented with records associated with each node. These records contain important semantic information that can be used during the decision process. Furthermore, since

```
                          (SENT A1)

         (NP A2)                   (VP *1*)      (.)

       (NOUN A2)                   (VP *2*)       .

      (NOUNP A2)                   (VP *3*)

      (WORD A2)                    (VP *4*)

  (#) (OUTPUT A2)       (VP *5*)              (NP A3)

   #     JOHN      (VERB *5*)                 (NP *7*)

                  (VERBP *6*)       (DETR *8*)       (NP *9*)

               (WORD *6*)    (S)    (WORD *10*)     (NOUN *9*)

           (#) (OUTPUT *6*) S   (#) (OUTPUT *10*) (NOUNP *9*)

            #      BUY           #        A         (WORD *9*)

                                                  (#) (OUTPUT *9*)

                                                   #       KITE
```

Figure 4-11: The Generated Tree

Paul is an utterance realization system, most of the difficult decisions are not of issue here. *Paul* does not have to try to find a tree structure that will fit a given sentence, as do parsing systems, nor does it have to attempt the selection and ordering of sentences to convey a desired message, as do utterance planning systems, and these are where the difficult decisions tend to lie.

Additionally, *Paul* exhibits the constraint of *locality* [32, 34]. Each decision can only make reference to information which is local to it. The system is not allowed to search through the existing tree for desired information. Relevant information must be explicitly passed on through local variables. A distinction between the use of locality here and its use in the MUMBLE system [32] is that in MUMBLE, *physical* locality was used, whereas in *Paul, conceptual* locality is used. Rather than using a node's position in a tree to determine what information is local to it, *Paul* uses the records associated with the tree nodes. These records contain the information local to their nodes. Nodes that represent more general structures have a wider scope of locality. For instance, the root node (representing the entire sentence) has all the semantic information known about the sentence local to it. As an example, assume we wanted to have a rule for sentences starting with subordinate clauses such that if the subject of the subordinate clause is the same as that for the main clause, the element should be pronominalized in the subordinate clause.

Because he doesn't like dogs, Bill kicked Carol's puppy.

Because MUMBLE depends on physical locality, it could not perform this rule unless it was explicitly stated to do so in the message. At the time the first reference to Bill is to be made in the subordinate clause, only those items that are physically located near this node in the tree are accessible. The subject of the main clause is not, and the decision whether to pronominalize based on this rule cannot be made. The only way MUMBLE could perform this task would be if the message explicitly stated that the subject of the subordinate clause were available for pronominalization. This would mean that the decision were no longer in the utterance realization stage, but forced upon the utterance planning stage.

Paul, on the other hand, uses conceptual locality. Before the tree is split up into the subordinate clause and the main clause, all the information that is local to the abstract node representing the sentence is available. This is true because the record representing the entire semantic information for the sentence already exists and is associated with this sentence node. (This would also be true with MUMBLE's message if it weren't processed strictly sequentially.) Therefore, it is an easy matter to check the element that will become the subject of the subordinate clause and compare it with that of the main clause. If they are the same, the subordinate clause can be marked to pronominalize its subject, and the desired sentence will be generated. Thus *Paul* is able to keep the decision within the realm of utterance realization.

The fact that *Paul* has semantic records associated with each of its nodes as it builds the tree allows the system to avoid the necessity for the *constraint-precedes* stipulation that is required for MUMBLE. The constraint-precedes stipulation dictates that the enumeration order of a sentence must be such that any element that causes constraints on other elements must be realized first. *Paul* doesn't require this because the information that such constraints exist is *conceptually* local to the node at the level where the decision has to be made. Thus, the concept of indelibility is maintained without adding the burden of the constraint-precedes stipulation.

By following the constraints of indelibility and locality, *Paul* also has the feature of running in bounded time between each output token. The number of operations required on a record before it is realized by surface output is fixed, and bears no relationship to the final length of the output sentence. In our above example, each rule can be applied only once to a record or its direct descendants. No looping occurs because the rules were carefully defined to avoid recursion. Therefore, there is a maximum of 22 rules that can be applied to any given record before it is realized into surface output. This time bound, which is stronger than a linear time constraint,[10] reflects the intuition that the generation process should proceed at a constant rate.

In summary, the generation paradigm for *Paul* is a bounded time, left-right, top down generator using an augmented phrase structure grammar. A surface structure tree is created of syntactic nodes with associated records. These records provide conceptually local semantic information, and allow the process to be indelible without the constraint of the constraint-precedes stipulation. This allows the process to proceed with a bounded number of operations between each entry into the output stream that is independent of the length of the final sentence.

[10]The linear time constraint states that the entire sentence must be processed within a time proportional to the length of the sentence. Other than this, there is no restriction to the amount of time spent between the output of each token. The difference between linear time and bounded time between each token is most evident on structures like left-branching sentences. The linear time system would have a long period outputting nothing, and then it would output the entire left-branching structure at once. The bounded time system would output each token at a relatively steady rate.

5. An Example

To help make the ideas discussed so far more concrete, an example is provided. The following is an actual example of text generated by *Paul*. In order to clearly demonstrate the system's ability at lexical substitution, the text to be generated should contain numerous references to various entities, both animate and inanimate. Therefore, *Paul* generates so-called children's stories, rather than something of more immediate applicability, such as explanation generation for an expert system or business letter generation, as was the original intention for *Epistle*. Unfortunately, these media generally do not offer the wealth of references to entities that is desired. Business letters typically refer to only the author of the letter, the recipient, the companies they respectively represent, and possibly some items sold by one or the other company. Justifications for expert systems are obviously restricted to the domain of expertise for the system. The explanations basically consist of causal links that form the knowledge of the system. Neither form of text offers the opportunity to describe several entities in varying manners, the way children's stories do. Therefore, the children's story is the most appropriate form of text for *Paul* to generate in order to demonstrate the full extent of its capabilities in lexical substitution.

The example discussed here is one about characters from Walt Kelly's *Pogo* comic strips. Of the characters mentioned in this example, Pogo is a male possum, Churchy is a male turtle, and Hepzibah is a female skunk.

Figure 5-1 contains the semantic representation for the example story to be generated, in the syntax of NLP records. After this comes Figure 5-2, showing the example story generated by *Paul* without any lexical substitution. While the version of the story in Figure 5-2 would be unacceptable as the final product of a text generator, it is shown here so that the reader can more easily understand the story represented semantically in Figure 5-1.

Even though this story is without lexical substitution, some simple forms of cohesion are exhibited. Because synonym substitution is not one of *Paul's* options for lexical substitution, the system uses synonyms throughout generation. This is demonstrated in the first two sentences. Note in Figure 5-1 that while the first two sentences of the story have the same primitive action as their heads ('like' in records **a1** and **b1**), they are realized by different words, "cares for" in the first sentence, and "likes" in the second sentence. This also shows that *Paul* takes advantage of the fact that synonyms exist in *all* parts of speech, not just for nouns. Additionally, when two consecutive sentences have the same primitive action as their heads, the system checks to see if any of the thematic roles, agent, affected, recipient, and attribute, are filled by the same entity. If any are, the word "too" is appended to the end of the second sentence, as the example demonstrates.

```
a1 ('like',exp: = 'a2',recip: = 'a3',stative);
a2 ('pogo');
a3 ('hepzibah');

b1 ('like',exp: = 'b2',recip: = 'a3',stative);
b2 ('churchy');

c1 ('give',agnt: = 'a2',aff: = 'c2',recip: = 'a3',active,effect: = 'c3');
c2 ('rose');
c3 ('enjoy\',recip: = 'a3',stative);

d1 ('want\',exp: = 'a3',recip: = 'd2',neg,stative);
d2 ('rose',possess: = 'b2');

e1 ('b2',char: = 'jealous',entity);

f1 ('hit\',agnt: = 'b2',aff: = 'a2',active);


g1 ('give',agnt: = 'b2',aff: = 'g2',recip: = 'a3',active);
g2 ('rose');

h1 ('drop\',exp: = 'h2',stative);
h2 ('petal',partof: = 'g2',plur);

i1 ('upset\',recip: = 'a3',cause: = 'h1',stative);

j1 ('cry\',agnt: = 'a3',active)[]
```

Figure 5-1: NLP Records for Example Story

POGO CARES FOR HEPZIBAH. CHURCHY LIKES HEPZIBAH, TOO. POGO GIVES A ROSE TO HEPZIBAH, WHICH PLEASES HEPZIBAH. HEPZIBAH DOES NOT WANT CHURCHY'S ROSE. CHURCHY IS JEALOUS. CHURCHY HITS POGO. CHURCHY GIVES A ROSE TO HEPZIBAH. PETALS DROP OFF. THIS UPSETS HEPZIBAH. HEPZIBAH CRIES.

Figure 5-2: Example Story without Lexical Substitution

Figure 5-3 is the story generated with pronoun substitution indiscriminately performed, and Figure 5-4 is the same with superordinate substitution. Just as with Figure 5-2, these versions are not acceptable text, and should not be mistaken to be the final output of *Paul*. Rather, they are presented here to dramatize the effects uncontrolled lexical substitution can have.

POGO LIKES HEPZIBAH. CHURCHY CARES FOR HER, TOO. HE GIVES A ROSE TO HER, WHICH PLEASES HER. SHE DOES NOT WANT HIS ROSE. HE IS JEALOUS. HE SLUGS HIM. HE GIVES A ROSE TO HER. PETALS DROP OFF. THIS UPSETS HER. SHE CRIES.

Figure 5-3: Example Story with Uncontrolled Pronoun Substitution

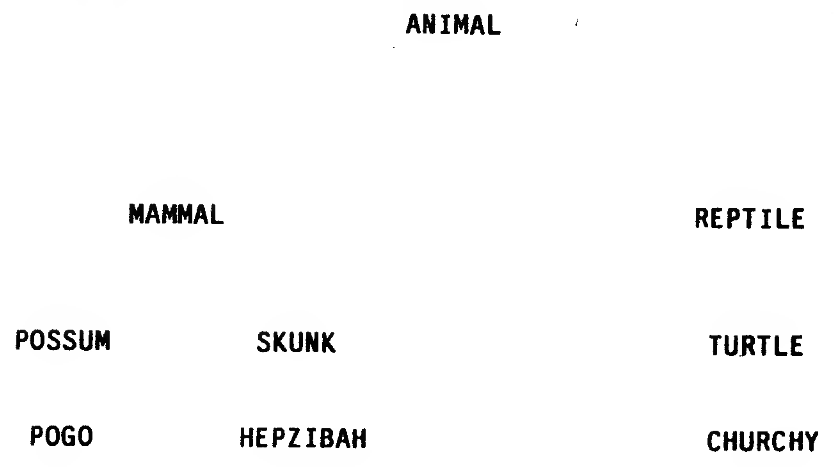
POGO LIKES HEPZIBAH. CHURCHY CARES FOR THE FEMALE ANIMAL, TOO. THE POSSUM GIVES A ROSE TO THE SKUNK, WHICH PLEASES THE BLACK MAMMAL. THE BLACK ANIMAL DOES NOT WANT THE REPTILE'S ROSE. THE TURTLE IS JEALOUS. THE SCALED ANIMAL PUNCHES THE MALE MAMMAL. THE REPTILE GIVES A ROSE TO THE SKUNK. PETALS FALL OFF. THIS UPSETS THE FEMALE MAMMAL. THE BLACK ANIMAL WEEPS.

Figure 5-4: Example Story with Uncontrolled Superordinate Substitution

For superordinate substitution, *Paul* assumes that its hierarchical database about the characters is common knowledge. Since this might not be true for all readers in this case, Figure 5-5 gives the pertinent information.

Once the system has determined that a superordinate substitution is to be made, several tasks must be accomplished. First of all, the superordinate must be selected for the referent. *Paul* searches up the hierarchical chain from the original record, making a list of all the records that are encountered along the way. For most types of entities, the chain stops with the record **THING**. However, this is not always the appropriate place to stop. Often going that far will produce superordinate substitutions that will sound insulting. For instance, referring to one's brother as "the boy" is not the same as referring to him as "the animal" or "the thing." The distinction seems to be that important attributes are lost with the last two, leaving the reader with the impression that these attributes are not to be found in the brother. The distinguishing attribute in the Pogo World is intelligence, and it is assumed that all animals in this world are indeed intelligent. Therefore, when going through the hierarchical chain, *Paul* will not go past the last attribute that has or can inherit the intelligence attribute.

After the list of acceptable superordinates has been created, one is selected randomly. Now *Paul* checks this superordinate against the other entities that have been used to date in the text. If none of the other entities are members of this superordinate set, or *superset* [13], the reference is unambiguous as it stands, and



1. POGO IS A MALE POSSUM.
2. HEPZIBAH IS A FEMALE SKUNK.
3. CHURCHY IS A MALE TURTLE.
4. POSSUMS ARE SMALL, GREY MAMMALS.
5. SKUNKS ARE SMALL, BLACK MAMMALS.
6. TURTLES ARE SMALL, GREEN REPTILES.
7. MAMMALS ARE FURRY ANIMALS.
8. REPTILES ARE SCALED ANIMALS.

Figure 5-5: The World in which the Example Story Exists

it is generated without modification. The first clause in the third sentence of Figure 5-4, *THE POSSUM GIVES A ROSE TO THE SKUNK...*, is an example of this. Since this world contains only one possum, Pogo, and only one skunk, Hepzibah, these superordinates can only refer to them, and *Paul* has generated them with no attempt to further disambiguate them.

However, if it turns out that entities other than the focal point being replaced are members of the chosen superset, the substitution must be modified to disambiguate the reference. *Paul* achieves this by selecting a physical attribute of the entity to be used as a modifier of the superordinate. The physical attributes that *Paul* looks for in the Pogo World are gender, color, size, and skin (furry, scaled, or feathered).

One of these attributes is randomly selected, and the focal point's inherited value for this attribute is generated as an adjective before the superordinate. The second phrase of the third sentence in Figure 5-4, ...*WHICH PLEASES THE BLACK MAMMAL* is an example of this. After MAMMAL has been selected as a superordinate substitution for Hepzibah, the system checked the remaining entities mentioned in the discourse so far. These were Pogo, Churchy, and a rose. Of the three, Pogo is a member of the Mammal superset, so the reference must be made unambiguous. The color attribute is randomly selected, and Hepzibah inherits the value black for this attribute. The system then generates the modified, and now unambiguous, noun phrase.

There is a problem, though, in that the attribute selected may not disambiguate the superordinate. For instance, what would have happened if, instead of selecting color as the disambiguating attribute, the system had chosen size? Rather than generating *THE BLACK MAMMAL*, the phrase *THE SMALL MAMMAL* would have been produced. Since Pogo is small, he is also a small mammal, and the modifying attribute has done nothing to disambiguate the superordinate. Similarly, a choice of skin as the modifying attribute would have led to the generation of *THE FURRY MAMMAL*, which is not only still ambiguous, but redundant, since in this world *all* mammals are furry. *Paul* avoids this problem by testing the inherited value for the selected attribute before generating it. If any of the previously mentioned entities that are members of the superset have the same value, this attribute is rejected, and another one is selected. This insures that the final result will be an unambiguous superordinate substitution.

Figure 5-6 is the example story without any lexical substitution again, but with each sentence's focus or expected focus list, obtained through the use of the Sidner algorithm. Now we are ready to follow *Paul* in the generation of this story with lexical substitution.

We start by initializing all the control variables to NIL. Then the first sentence is generated. Because there are no previously generated references, there can be no focal points, and the sentence *Pogo cares for Hepzibah*, is out put. Additionally, the relevant facts about these references are stored, as shown in Figure 5-7.

When the next sentence is generated, the first noun phrase encountered is *Churchy*. Since this is not a member of the list of noun phrases mentioned in the text, it is not a focal point, and not subject to lexical substitution. When the system comes to *Hepzibah*, however, we *do* have a focal point, since *Hepzibah* is a member of the MENTIONED IN THE TEXT list, and *Paul* must determine the class of this focal point. Since *Hepzibah* is the last female to have been mentioned within an acceptable distance, the focal point is Class I. (Note that because *Hepzibah* was also the focus of the previous sentence, this would also make it a Class I focal point.) Therefore, a pronoun substitution is required.

1. POGO LIKES HEPZIBAH.

expected focus list: *"Hepzibah", "Pogo"*

2. CHURCHY LIKES HEPZIBAH, TOO.

expected focus list: *"Hepzibah", "Churchy"*

3. POGO GIVES A ROSE TO HEPZIBAH,

expected focus list: *"a rose", "Hepzibah", "Pogo"*

4. WHICH PLEASES HEPZIBAH.

expected focus list: *"Hepzibah", "Pogo gives a rose to Hepzibah"*

5. HEPZIBAH DOES NOT WANT CHURCHY'S ROSE.

expected focus list: *"Churchy's rose", "Hepzibah"*

6. CHURCHY IS JEALOUS.

focus: *"Churchy"*

7. CHURCHY HITS POGO.

expected focus list: *"Pogo", "Churchy"*

8. CHURCHY GIVES A ROSE TO HEPZIBAH.

expected focus list: *"a rose", "Hepzibah", "Churchy"*

9. PETALS FALL OFF.

focus: *"petals"*

10. THIS UPSETS HEPZIBAH.

expected focus list: *"Hepzibah", "Petals fall off."*

11. HEPZIBAH CRIES.

focus: *"Hepzibah"*

Figure 5-6: Expected Focus Lists

With the third sentence, we have have three entities, *Pogo, Hepzibah,* and a rose, which we will refer to as *rose1*. Of these, *Pogo* and *Hepzibah* are focal points. *Hepzibah* is still the only female mentioned within two sentences, and is still the focal point of the previous sentence, so it is still a Class I focal point, subject to pronominalization. *Pogo*, however, is neither of these, and is not Class I. Nor was it the last male mentioned, so the focal point is not Class II. Similarly, *Pogo* fails the criteria for Class III, leaving us with Class IV, and allowing only a definite noun phrase to be generated. In the case of proper nouns, they already are definite

```
SENTENCE GENERATED: POGO CARES FOR HEPZIBAH.

LAST MALES: POGO NIL                LAST FEMALES: HEPZIBAH NIL
LAST NEUTERS: NIL NIL               LAST PLURALS: NIL NIL

AGENT: POGO                         AFFECTED: NIL
RECIPIENT: HEPZIBAH                 ATTRIBUTE: NIL

FOCUS: HEPZIBAH

MENTIONED LAST SENTENCE: NIL
MENTIONED THIS SENTENCE: POGO HEPZIBAH

MENTIONED IN THE TEXT: POGO HEPZIBAH
```

Figure 5-7: Control Variables After First Sentence

```
SENTENCE GENERATED: CHURCHY LIKES HER, TOO.

LAST MALES: CHURCHY POGO            LAST FEMALES: HEPZIBAH HEPZIBAH
LAST NEUTERS: NIL NIL               LAST PLURALS: NIL NIL

AGENT: CHURCHY                      AFFECTED: NIL
RECIPIENT: HEPZIBAH                 ATTRIBUTE: NIL

FOCUS: HEPZIBAH

MENTIONED LAST SENTENCE: POGO HEPZIBAH
MENTIONED THIS SENTENCE: CHURCHY HEPZIBAH

MENTIONED IN THE TEXT: POGO HEPZIBAH CHURCHY
```

Figure 5-8: Control Variables After Second Sentence

noun phrases, so "Pogo" is simply generated.

SENTENCE GENERATED: **POGO GIVES A ROSE TO HER.**

LAST MALES: *POGO CHURCHY*
LAST FEMALES: *HEPZIBAH HEPZIBAH*
LAST NEUTERS: *ROSE1 NIL*
LAST PLURALS: *NIL NIL*

AGENT: *POGO*
AFFECTED: *ROSE1*
RECIPIENT: *HEPZIBAH*
ATTRIBUTE: *NIL*

FOCUS: *ROSE1*

MENTIONED LAST SENTENCE: *CHURCHY HEPZIBAH*
MENTIONED THIS SENTENCE: *POGO ROSE1 HEPZIBAH*

MENTIONED IN THE TEXT: *POGO HEPZIBAH CHURCHY ROSE1*

Figure 5-9: Control Variables After Third Sentence

The second clause of this sentence (which for our purposes constitutes a distinct sentence) starts with a relative pronoun. When a record has an effect attribute (such as **c1** in Figure 5-1) which is itself another clause (such as **c3** in Figure 5-1), the second clause is generated as a relative clause, and the result is in Figure 5-10.

The details of the next few sentences are similar enough that it would be worth our while to skip forward a bit to the ninth sentence. The only entity mentioned in this sentence is *petals*, which have not yet been mentioned in the discourse. However, if we look at record **h2** in Figure 5-1, we see that these petals are part of the rose from sentence 8. Therefore, we are not really referring to a new entity, but rather a part of an old one, and our generated text should make this clear. We actually do have a focal point, even though the entity is not on the **MENTIONED IN THE TEXT** list after the previous sentence. In order to determine if an entity is a Class V focal point, *Paul* checks each member of the **MENTIONED IN THE TEXT** list. If the item is a part of one of the members, we genuinely do have a Class V focal point. If not, then we simply have an item that is being mentioned for the first time, and it can be treated in the usual fashion. Figure 5-11 demonstrates how the Class V focal point was handled in this specific example.

SENTENCE GENERATED: **WHICH PLEASES HER.**

LAST MALES: *NIL POGO*
LAST NEUTERS: *NIL ROSE1*

LAST FEMALES: *HEPZIBAH HEPZIBAH*
LAST PLURALS: *NIL NIL*

AGENT: *NIL*
RECIPIENT: *HEPZIBAH*

AFFECTED: *NIL*
ATTRIBUTE: *NIL*

FOCUS: *HEPZIBAH*

MENTIONED LAST SENTENCE: *POGO ROSE1 HEPZIBAH*
MENTIONED THIS SENTENCE: *HEPZIBAH*

MENTIONED IN THE TEXT: *POGO HEPZIBAH CHURCHY ROSE1*

Figure 5-10: Control Variables After Fourth Sentence

SENTENCE GENERATED: **THE PETALS DROP OFF.**

LAST MALES: *NIL CHURCHY*
LAST NEUTERS: *NIL ROSE3*

LAST FEMALES: *NIL HEPZIBAH*
LAST PLURALS: *PETALS NIL*

AGENT: *PETALS*
RECIPIENT: *NIL*

AFFECTED: *NIL*
ATTRIBUTE: *NIL*

FOCUS: *PETALS*

MENTIONED LAST SENTENCE: *CHURCHY ROSE3 HEPZIBAH*
MENTIONED THIS SENTENCE: *PETALS*

MENTIONED IN THE TEXT:
POGO HEPZIBAH CHURCHY ROSE1 ROSE2 ROSE3 PETALS

Figure 5-11: Control Variables After Ninth Sentence

This should be sufficient to provide the reader with an understanding of how lexical substitution is controlled in *Paul.* The "snapshots" of the control variables after each sentence can be found in Appendix II. The final complete text for this story is:

> POGO CARES FOR HEPZIBAH. CHURCHY LIKES HER, TOO. POGO GIVES A
> ROSE TO HER, WHICH PLEASES HER. SHE DOES NOT WANT CHURCHY'S ROSE.
> HE IS JEALOUS. HE PUNCHES POGO. HE GIVES A ROSE TO HEPZIBAH.
> THE PETALS FALL OFF. THIS UPSETS HER. SHE CRIES.

Appendix III contains additional examples of text generated by *Paul.*

6. Related Work

While natural language processing has been a subject of investigation for decades, text generation has only recently enjoyed serious research endeavors [30]. That is not to say that work in text generation did not exist before a few years ago. In 1969, Harper and Su reported a system that composed paragraphs in Russian on topics in the domain of physics [12]. The system was designed to demonstrate the development of the chosen theme and exhibit cohesion between the generated sentences. All this was achieved with the random selection of constituents.

The system first randomly chooses a syntactic sentence pattern which has restrictions as to what can appear in each of its slots. Then based on those restrictions, words of the proper syntactic categories are randomly selected to fill the slots and create a sentence. The weights of elements for future random selections (both sentence patterns and words) are altered based on previous selections. This way, the system favors constructions that the authors feel reflect development of theme and cohesion between the sentences. The words for this system are arranged into syntactic classes. Additionally, some semantic information is stored in that each word entry has pointers to other words that are synonyms, antonyms, and superordinates.

An important shortcoming of this system is that it has no semantic representation of what has been or should be said. After randomly selecting a sentence pattern, words are randomly selected and fitted into this pattern. Cohesion is attempted by weighting random selections to favor those words and constructions that seem to provide reference preference, and style isn't considered at all. While the system's dictionary is arranged in a superordinate hierarchy, the semantic information the dictionary contains is very limited and inadequate. The entries consist of the words themselves, rather than the conceptual actions and objects these words represent. This approach violates almost all of the six criteria for natural language generation. While it is an important first step in the field, it is impractical for further development.

The HAM-RPM system [44] is an interesting advance in natural language generation. HAM-RPM was designed to be a question-answer system about visible scenes. Given an appropriate internal representation of some scene, its objects, and their spatial relationships to each other, HAM-RPM will answer questions about this scene. For its generation component, the research emphasis was on noun phrase generation, specifically, the generation of noun phrases that would not be ambiguous to a human witnessing the scene.

HAM-RPM was designed to give single sentence responses to queries. Therefore, most of the issues of cohesion generally don't apply to the problem the system addresses. What cohesion it did express was only for noun phrases, and was heavily based on spatial relationships, which is part of exophoric reference. However, exophoric reference can only be used in conversational applications, where both parties are present and witnessing the same scene. In a context such as business letters or medical diagnosis based on test results, exophoric reference cannot be used, and *endophoric* reference [11] must be used to achieve cohesion. HAM-RPM is an interesting system, but it addressed a problem that is significantly different from the problem *Paul* attempts to address, and the approach of HAM-RPM is incompatible with *Paul.*

Since HAM-RPM, Jameson and Wahlster have reported the development of HAM-ANS [18], a dialogue system designed to employ a user model in anaphora generation. The system is a question/answering system in which the program plays the part of a hotel clerk answering questions about available rooms. In order to make the responses seem more natural, a capability for anaphora in the form of ellipsis and definite description has been incorporated into the system. Before an elliptic response is generated, the proposed answer is passed back to the system's parser by what is known as an *anticipation feedback loop.* If the response can be unambiguously parsed, it is actually given as output. If, however, the response proves to be ambiguous, a less elliptic response is created and fed to the feedback loop. This way, the system ensures that the user will not be confused by ambiguous answers. The generation of definite description is based on both the occurrence of previous references to the object in question, and a desire of the system (in its role as hotel clerk) to describe the available room to the user (in her role as potential customer) in a manner designed to maximize the apparent desirability of the room, based on the system's model of the particular user.

HAM-ANS is once again a strictly conversational language generator. The ellipsis it employs is not one designed to avoid tedious repetition, as is proposed by the syntactic transformation of Equi NP Deletion [1]. Rather, this ellipsis reflects the natural tendency of people to use incomplete, though acceptable and unambiguous, sentences. The use of definite description that the system demonstrates also does not have cohesion as its main goal. The user model gives the system a basis in order to describe hotel rooms in the best possible light according to a specific user model. Instead of trying to be completely clear and unambiguous, the system will often use a definite description where one isn't appropriate or deliberately not use one where it should be so that the user can be misled without the system actually lying. Of course, this is not a linguistic phenomenon, but a psychological one that employs language.

CES [26] is another system that attempted some cohesion in generation. CES recognized that text generation consists of the two subtasks mentioned above, and the authors chose to concentrate mainly on the first one of utterance planning. The system works very hard at determining the minimum that is required to be said and still be unambiguously clear. This is achieved by giving the system a representation of the context in which a single sentence is to be generated. While only single sentences are being generated, by being in a context the sentences can exhibit cohesion suitable for that context.

Unfortunately, the only cohesive devices that were explored at all were pronominalization and ellipsis. While context is used to achieve some cohesion, no stylistic considerations are made. Furthermore, the system has been only partially implemented.

The GEN system [22] divides text generation into three subtasks. The first is to create the knowledge structures representing what is to be said. (Katz calls these structures kernel phrase markers.) The second is to determine which linguistic transformations [1] are to be performed on the kernels, based on syntactic and thematic considerations. The final step, which is the one GEN is designed to perform, is to perform the transformations specified in the second step and translate the transformed kernel phrase markers into the target natural language. It is assumed that all semantic and pragmatic knowledge is represented as a set of frames.

Because GEN is heavily based on the syntactic aspects of transformational grammar, it exhibits all the limitations of this approach to linguistics. Transformational grammar is designed to take as input a syntactic tree, representing the deep structure of a sentence, to perform syntactic transformations on this tree, and translate the transformed tree into a surface sentence. This is exactly the approach GEN takes. Therefore, little semantic knowledge is incorporated. Cohesion is shown only through the use of pronouns, and only one rule of pronominalization is employed.

Another system based on transformational grammar is the transformational generator described in [2]. Designed to generate examples of good English as an aid in teaching the deaf and learners of English as a second language, the generator is divided into three parts. First, a set of context free rules, called the base component, creates a tree structure. A transformer then applies transformational rules to the trees to derive a surface tree. A multilevel control mechanism helps constrain the tasks of the other two components. A dictionary and semantic network prevent the generation of syntactically correct but semantically meaningless sentences, such as "Colorless green ideas sleep furiously."

As with GEN, the transformational generator has severe limitations in semantic applications. The emphasis on this work is in generating grammatically correct English sentences, and the semantic meaning behind those sentences is completely ignored. Furthermore, isolated sentences are once again being generated, so the problems of cohesion and style, which are more important for multisentential text, have not been addressed.

The XPLAIN system [43] proposed a solution to a significant problem to text generation as it is applied to expert systems. It has long been recognized that expert systems must be able to explain their conclusions and how these conclusions were derived. However, XPLAIN realized that in addition, expert systems need to justify their methods for arriving at their conclusions, rather than merely giving these methods as an explanation. This was achieved by having the system generate its own rules and then applying those rules to specific cases. In essence, in addition to knowing what to do in a given situation, XPLAIN knows why it should be done.

Swartout states that the focus of XPLAIN is in utterance planning based on the information the system has behind its rule base creation, although some cohesive devices are used in the system. Relative clauses can be created to describe causal chains. However, these devices are used only in specific circumstances in a very *ad hoc* manner. Style is addressed to some extent in that the system is able to generate explanations at different levels of complexity. But this is done strictly by first generating different knowledge structures at different levels of complexity. Once the structures are created, no further consideration is given to the impact of vocabulary selection on style.

KDS [28, 29] is a recent system that proposes a new paradigm for natural language generation. This approach, called the fragment-and-compose paradigm, takes a semantic data structure, fragments it into little pieces, each of which could represent a simple sentence, and composes full sentences and paragraphs from these pieces. The system selects from all the myriad ways of expressing a concept by creating each of the possible abstract representations (which the authors call *protosentences*), and evaluating each one. Eventually, a final set of protosentences is created and fed to a generator. The generator produces sentences one at a time with very little consideration of the previously produced sentences.

Obviously, the center of research for KDS is utterance planning, and once again the authors admit little work in utterance realization. Some cohesion is achieved through pronouns and incomplete descriptions, but the possibilities have not been fully explored. While one of the modules of KDS is responsible for selecting a text presentation style and organizing the fragmented pieces into a text content consistent with the selected style, no consideration is given to style during the actual generation.

One system that addressed the problems of utterance realization is the MUMBLE system [32, 34]. MUMBLE again divides the natural language task into two separate subtasks. An expert program, known as the *speaker*, performs the subtask of utterance planning by creating *messages* representing what needs to be said, and the generator, consisting of a *dictionary* and a *linguistic component*, turns these sentences into surface English. The representation of these messages should be determined by the speaker, *not* the linguistic component. The linguistic component should be able to accept and process messages in the representation that is most natural for the domain of the speaker, rather than dictating an arbitrary representation for all speakers in all domains. The dictionary is the component that contains the knowledge for translating the domain specific representation into English, and therefore each representation must have its own compiled dictionary. This scheme allows the generator to be driven by the goals of the speaker, rather than by the structure of the grammar. The speaker is able to state explicit goals (such as emphasizing a specific point or contrasting two items) in its messages, and the linguistic component is driven to achieve these goals. Additionally, MUMBLE exhibits several constraints included to provide both efficiency in generation and a theory that was grounded on psycholinguistically plausible hypotheses. These constraints include *indelibility, locality,* and *running in bounded time,* as they have been discussed in Chapter 4.

The chief emphasis of MUMBLE was the idea of driving the linguistic component by the explicitly stated goals of the expert system speaker. Cohesive devices were used as one of the means for achieving these goals, but they were not the central issue of the work. Nor were they used to specifically perform the two major tasks of cohesive devices, the avoidance of boring redundancy and the distinction of new information from old. Furthermore, no theory was offered to provide control over the use of cohesive devices such as lexical substitution.

The differences between MUMBLE and *Paul* concerning their uses of computational constraints have already been discussed in Chapter 4. To briefly recap, by using an augmented phrase rule grammar, *Paul* is able to maintain the constraints of indelibility and locality. Furthermore, the notion of locality has been extended to define locality as *conceptual* instead of *physical.* This, with the proper use of augmented phrase structure rules, frees *Paul* from some of the limitations exhibited in MUMBLE, such as the constraint-precedes stipulation.

A recent system that made significant progress towards fulfilling the six criteria for natural language generation is TEXT [35]. In this work, the language generation process is again divided into the utterance planning and the utterance realization tasks. TEXT addresses the problems of what to say and how to organize it effectively, using the principles of discourse structure, discourse coherency, and relevancy criteria. The system was developed to respond to data base queries. Once a question is received, a *relevant knowledge pool* is constructed. This is a subset of the data base and contains all the information that can be included in the response. Next a *schema* is selected, based on the type of question and the information in the relevant knowledge pool. The schema dictates the structure of the response, and is used to determine the order in which the sentences are to be generated. Finally, focus is used to obtain an overall coherency in the generated text.

A significant difference between TEXT and *Paul* is their use of focus. TEXT uses focus for coherency in text, making the text seem to have a logical flow in it. *Paul* uses focus with other factors to achieve cohesiveness in the text, making the sentences of the text to be interconnected and part of a larger unit. Furthermore, the emphasis on TEXT is again in utterance planning and the problems of that realm. Cohesive devices such as lexical substitution are used to achieve the goals of the specific schema and the general ones of controlling focus. The utterance realization aspects of cohesion, of avoiding redundancy and marking of new information through the controlled use of lexical substitution, are not discussed.

In conclusion, none of the above systems addresses all six of the criteria necessary for good natural language generation. This is true because for the most part these systems have focused on utterance planning rather than on utterance realization and the problems associated with this task. In particular, none of these systems address the problem of cohesion in a methodical manner. As we have seen, *Paul* is a system that specifically addresses the utterance realization problem of cohesion by presenting an orderly approach to lexical substitution.

7. Conclusions

7.1. Contributions of *Paul*

Paul is one of the few text generation systems designed specifically to address issues of utterance realization. As such, several advances in the field were made with this work.

1. This is the first system to perform a full range of lexical substitutions. No other existing system offers synonymous substitution, superordinate substitution, pronominalization, and definite noun phrases. This was achieved by identifying the *minimal features* of the elements, and determining the least amount of information required to generate unambiguous references.

2. *Paul* is the first system that offers a theory for controlling the selection of lexical substitutions. This theory identifies five classes of potential antecedence, and associates a strength of antecedence recovery with each type of lexical substitution. *Paul* is capable of determining the potential antecedence class for each element in the discourse, and selecting the appropriate lexical substitution based on this class.

3. *Paul* is able to use these lexical substitutions to generate cohesive text. Specifically, *Paul* avoids unnecessary repetition and marks old information from new by the judicious application of lexical substitutions. These functions are required before a passage can be recognized as text.

4. *Paul* uses augmented phrase structure rules to achieve indelibility in generation. By associating records of semantic information with each node in the syntactic structure tree as it is being created, decisions can be confidently made that would otherwise require backtracking or expansion in parallel.

5. Augmented phrase structure rules are also used in *Paul* to fulfill the constraint of conceptual locality. In order to avoid searches throughout the entire syntactic structure tree (which might not completely exist at the time of the search), the locality constraint dictates that a decision at a node can only use information local to that node. But rather than defining local information as that physically near in the tree, *Paul* defines local information conceptually, through the use of the semantic records associated with each node. This way, locality is achieved without further constraints.

6. *Paul* is able to run in bounded time. There are a fixed number of steps that will be taken before the next word is generated. This was achieved because *Paul* is indelible and follows the locality constraint, while avoiding recursive rules through careful application of condition and creation specifications.

7.2. Limitations of *Paul*

No program can do everything, and *Paul* is certainly no exception to this rule. There are several limitations exhibited in *Paul*.

1. *Paul* performs utterance realization *only*. It is completely incapable of performing utterance planning tasks. The system takes semantic records of what to say as input. Currently these records have to be created by hand.

2. Of the many cohesive devices discussed in this report, *Paul* only performs those of lexical substitution. Ellipsis, conjunction, reference and substitution are beyond the ken of this system.

3. *Paul* assumes an endophoric context when selecting cohesive devices. The system cannot correctly generate exophoric references. It cannot talk about its world.

4. The system does not have a user model of the reader's knowledge and beliefs. *Paul* currently assumes that the user knows what it knows, that the facts in its data base are common knowledge.

7.3. Future Research

1. One important issue not addressed in *Paul* is the question of style. Especially when the applications of text generators move toward more serious fields, such as expert systems explanations and justifications, and business correspondence, it will be necessary to be able to vary the style and mood of the text being generated. As we have seen with general nouns, lexical substitution can have a very great impact on the style, and maintaining a specific style will add unexplored constraints on the lexical substitution selection process.

2. As we have stated above, the theory used for controlling lexical substitution in *Paul* has been applied *only* to lexical substitution. It remains to be seen if this theory can be extended to control the selection of other cohesive devices, and whether a general theory can be found to control *all* cohesive devices. This extension would hopefully include exophoric reference. This would require the program having a sense of the "world" it "exists" in.

3. *Paul* only generates texts of single paragraph size, and the cohesive devices discussed apply to binding sentences together within that paragraph. The issues of multi-paragraph text generation remain to be researched. Are the cohesive devices that tie sentences together be used to associate paragraphs? Are there other cohesive devices that are used only to bind paragraphs? Do paragraphs have an ordered surface structure, the way sentences have? These questions remain to be answered.

In conclusion, the field of text generation, and especially the branch dealing with utterance realization, is rich with interesting topics to explore. With the growing necessity for expert systems to be able to explain themselves, and the increasing demand for programs with human factor considerations, the need for good text generators is one of the most dynamically expanding fields of artificial intelligence.

APPENDIX I

from *Alice's Adventures in Wonderland* by Lewis Carroll

The White Rabbit put on his spectacles. 'Where shall I begin, please your Majesty?' he asked.

'Begin at the beginning,' the King said, very gravely, 'and go on till you come to the end: then stop.'

There was a dead silence in the court, whilst the White rabbit read out these verses:

'They told me you had been to her,
And mentioned me to him:
She gave me a good character,
But said I could not swim.

He sent them word I had not gone
(We know it to be true):
If she should push the matter on,
What would become of you?

I gave her one, they gave him two,
You gave us three or more;
They all returned from him to you,
Though they were mine before.

If I or she should chance to be
Involved in this affair,
He trusts to you to set them free,
Exactly as we were.

My notion was that you had been
(Before she had this fit)
An obstacle that came between
Him, and ourselves, and it.

Don't let him know she liked them best,
For this must ever be
A secret, kept from all the rest,
Between yourself and me.'

'That's the most important piece of evidence we've heard yet,' said the King, rubbing his hands...

Appendix II

Trace of Control Variables for the Example Story

SENTENCE GENERATED: **POGO CARES FOR HEPZIBAH.**

LAST MALES: *POGO NIL*	LAST FEMALES: *HEPZIBAH NIL*
LAST NEUTERS: *NIL NIL*	LAST PLURALS: *NIL NIL*

AGENT: *POGO*	AFFECTED: *NIL*
RECIPIENT: *HEPZIBAH*	ATTRIBUTE: *NIL*

FOCUS: *HEPZIBAH*

MENTIONED LAST SENTENCE: *NIL*
MENTIONED THIS SENTENCE: *POGO HEPZIBAH*

MENTIONED IN THE TEXT: *POGO HEPZIBAH*

Control Variables After First Sentence

SENTENCE GENERATED: **CHURCHY LIKES HER, TOO.**

LAST MALES: *CHURCHY POGO*	LAST FEMALES: *HEPZIBAH HEPZIBAH*
LAST NEUTERS: *NIL NIL*	LAST PLURALS: *NIL NIL*

AGENT: *CHURCHY*	AFFECTED: *NIL*
RECIPIENT: *HEPZIBAH*	ATTRIBUTE: *NIL*

FOCUS: *HEPZIBAH*

MENTIONED LAST SENTENCE: *POGO HEPZIBAH*
MENTIONED THIS SENTENCE: *CHURCHY HEPZIBAH*

MENTIONED IN THE TEXT: *POGO HEPZIBAH CHURCHY*

Control Variables After Second Sentence

SENTENCE GENERATED: **POGO GIVES A ROSE TO HER,**

LAST MALES: *POGO CHURCHY*	LAST FEMALES: *HEPZIBAH HEPZIBAH*
LAST NEUTERS: *ROSE1 NIL*	LAST PLURALS: *NIL NIL*

AGENT: *POGO*	AFFECTED: *ROSE1*
RECIPIENT: *HEPZIBAH*	ATTRIBUTE: *NIL*

FOCUS: *ROSE1*

MENTIONED LAST SENTENCE: *CHURCHY HEPZIBAH*
MENTIONED THIS SENTENCE: *POGO ROSE1 HEPZIBAH*

MENTIONED IN THE TEXT: *POGO HEPZIBAH CHURCHY ROSE1*

Control Variables After Third Sentence

SENTENCE GENERATED: **WHICH PLEASES HER.**

LAST MALES: *NIL POGO*	LAST FEMALES: *HEPZIBAH HEPZIBAH*
LAST NEUTERS: *NIL ROSE1*	LAST PLURALS: *NIL NIL*

AGENT: *NIL*	AFFECTED: *NIL*
RECIPIENT: *HEPZIBAH*	ATTRIBUTE: *NIL*

FOCUS: *HEPZIBAH*

MENTIONED LAST SENTENCE: *POGO ROSE1 HEPZIBAH*
MENTIONED THIS SENTENCE: *HEPZIBAH*

MENTIONED IN THE TEXT: *POGO HEPZIBAH CHURCHY ROSE1*

Control Variables After Fourth Sentence

SENTENCE GENERATED: **SHE DOES NOT WANT CHURCHY'S ROSE.**

LAST MALES: *CHURCHY NIL* LAST FEMALES: *HEPZIBAH HEPZIBAH*
LAST NEUTERS: *ROSE2 NIL* LAST PLURALS: *NIL NIL*

AGENT: *HEPZIBAH* AFFECTED: *NIL*
RECIPIENT: *ROSE2* ATTRIBUTE: *NIL*

FOCUS: *ROSE2*

MENTIONED LAST SENTENCE: *HEPZIBAH*
MENTIONED THIS SENTENCE: *HEPZIBAH ROSE2 CHURCHY*

MENTIONED IN THE TEXT: *POGO HEPZIBAH CHURCHY ROSE1 ROSE2*

Control Variables After Fifth Sentence

SENTENCE GENERATED: **HE IS JEALOUS.**

LAST MALES: *CHURCHY CHURCHY* LAST FEMALES: *NIL HEPZIBAH*
LAST NEUTERS: *NIL ROSE2* LAST PLURALS: *NIL NIL*

AGENT: *CHURCHY* AFFECTED: *NIL*
RECIPIENT: *NIL* ATTRIBUTE: *NIL*

FOCUS: *CHURCHY*

MENTIONED LAST SENTENCE: *HEPZIBAH ROSE2 CHURCHY*
MENTIONED THIS SENTENCE: *CHURCHY*

MENTIONED IN THE TEXT: *POGO HEPZIBAH CHURCHY ROSE1 ROSE2*

Control Variables After Sixth Sentence

```
SENTENCE GENERATED: HE PUNCHES POGO.

LAST MALES: POGO CHURCHY                 LAST FEMALES: NIL NIL
LAST NEUTERS: NIL NIL                    LAST PLURALS: NIL NIL

AGENT: CHURCHY                           AFFECTED: POGO
RECIPIENT: NIL                           ATTRIBUTE: NIL

FOCUS: POGO

MENTIONED LAST SENTENCE: CHURCHY
MENTIONED THIS SENTENCE: CHURCHY POGO

MENTIONED IN THE TEXT: POGO HEPZIBAH CHURCHY ROSE1 ROSE2
```

Control Variables After Seventh Sentence

```
SENTENCE GENERATED: HE GIVES A ROSE TO HEPZIBAH.

LAST MALES: CHURCHY POGO                 LAST FEMALES: HEPZIBAH NIL
LAST NEUTERS: ROSE3 NIL                  LAST PLURALS: NIL NIL

AGENT: CHURCHY                           AFFECTED: ROSE3
RECIPIENT: HEPZIBAH                      ATTRIBUTE: NIL

FOCUS: ROSE3

MENTIONED LAST SENTENCE: CHURCHY POGO
MENTIONED THIS SENTENCE: CHURCHY ROSE3 HEPZIBAH

MENTIONED IN THE TEXT: POGO HEPZIBAH CHURCHY ROSE1 ROSE2 ROSE3
```

Control Variables After Eighth Sentence

SENTENCE GENERATED: **THE PETALS FALL OFF.**

LAST MALES: *NIL CHURCHY*
LAST FEMALES: *NIL HEPZIBAH*
LAST NEUTERS: *NIL ROSE3*
LAST PLURALS: *PETALS NIL*

AGENT: *PETALS*
AFFECTED: *NIL*
RECIPIENT: *NIL*
ATTRIBUTE: *NIL*

FOCUS: *PETALS*

MENTIONED LAST SENTENCE: *CHURCHY ROSE3 HEPZIBAH*
MENTIONED THIS SENTENCE: *PETALS*

MENTIONED IN THE TEXT:
POGO HEPZIBAH CHURCHY ROSE1 ROSE2 ROSE3 PETALS

Control Variables After Ninth Sentence

SENTENCE GENERATED: **THIS UPSETS HER.**

LAST MALES: *NIL NIL*
LAST FEMALES: *HEPZIBAH NIL*
LAST NEUTERS: *NIL NIL*
LAST PLURALS: *NIL PETALS*

AGENT: *NIL*
AFFECTED: *NIL*
RECIPIENT: *HEPZIBAH*
ATTRIBUTE: *NIL*

FOCUS: *HEPZIBAH*

MENTIONED LAST SENTENCE: *PETALS*
MENTIONED THIS SENTENCE: *HEPZIBAH*

MENTIONED IN THE TEXT:
POGO HEPZIBAH CHURCHY ROSE1 ROSE2 ROSE3 PETALS

Control Variables After Tenth Sentence

SENTENCE GENERATED: SHE **CRIES.**

LAST MALES: *NIL NIL*
LAST NEUTERS: *NIL NIL*

LAST FEMALES: *HEPZIBAH HEPZIBAH*
LAST PLURALS: *NIL NIL*

AGENT: *HEPZIBAH*
RECIPIENT: *NIL*

AFFECTED: *NIL*
ATTRIBUTE: *NIL*

FOCUS: *HEPZIBAH*

MENTIONED LAST SENTENCE: *HEPZIBAH*
MENTIONED THIS SENTENCE: *HEPZIBAH*

MENTIONED IN THE TEXT:
POGO HEPZIBAH CHURCHY ROSE1 ROSE2 ROSE3 PETALS

Control Variables After Eleventh Sentence

Appendix III

Additional Examples of Generated Stories

POGO CARES FOR HEPZIBAH. CHURCHY LIKES HER, TOO. POGO GIVES A ROSE TO HER, WHICH PLEASES HER. SHE DOES NOT WANT CHURCHY'S ROSE. HE IS JEALOUS. HE PUNCHES POGO. HE GIVES A ROSE TO HEPZIBAH. THE PETALS DROP OFF. THIS UPSETS HER. SHE CRIES.

CHURCHY LIKES HEPZIBAH. SHE DOES NOT CARE FOR HIM. THIS UPSETS HIM. HE KISSES HER. SHE WEEPS. THIS ANGERS POGO. HE HITS CHURCHY.

POGO AND CHURCHY GO TO THE STORE. CHURCHY PURCHASES A KITE. HE GIVES IT TO POGO. THE POSSUM GIVES IT TO HEPZIBAH, WHICH PLEASES HER. SHE KISSES HIM. THIS UPSETS CHURCHY. HE WEEPS.

CHURCHY AND POGO GO TO THE STORE. HEPZIBAH GOES, TOO. POGO BUYS A KITE. HE GIVES IT TO HER, WHICH PLEASES HER. SHE KISSES HIM. THIS ANGERS CHURCHY. HE TAKES THE KITE. HE BREAKS IT. THIS UPSETS HEPZIBAH. SHE CRIES. POGO SLUGS CHURCHY.

Appendix IV

BNF for NLP

```
<COMMAND> ::= <LEFT> --> <RIGHT> <END>

<LEFT> ::= <SEG-TYPE> | <SEG-TYPE> ( <TEST> )

<SEG-TYPE> ::= <IDENTIFIER>

<TEST> ::= <ATTR-CONDITION> | <TEST> , <TEST> | <TEST> <OR> <TEST> |
           ( <TEST> ) | ↑ <TEST>

<ATTR-CONDITION> ::= <ATTRIBUTE> <PV> | <VALUE> = <VALUE> |
                     ! <S-EXPRESSION> | <FUNCTION-CALL> |
                     <$REFERENCE> | ' <IDENTIFIER> '

<ATTRIBUTE> ::= <IDENTIFIER> | @[ <ATTRIBUTE> <PV> ] | @[ <$REFERENCE> ]

<PV> ::= <0> | ( <VALUE> )

<VALUE> ::= ' <IDENTIFIER> ' | <ATTRIBUTE> <PV> | <$REFERENCE>

<RIGHT> ::= <SEG-TYPE> | <SEG-TYPE> ( <CREATION> ) | <RIGHT> <RIGHT>

<CREATION> ::= <ATTR-CREATION> | <CREATION> , <CREATION>

<ATTR-CREATION> ::= <ATTRIBUTE> <PV> | <ATTRIBUTE> <PV> := <VALUE> |
                    <$REFERENCE> | % <ATTRIBUTE> <PV> |
                    - <ATTRIBUTE> <PV> | ' <IDENTIFIER> ' |
                    <ATTRIBUTE> <PV> := <$REFERENCE> |
                    <ATTRIBUTE> <PV> := <FUNCTION-CALL> |
                    ! <S-EXPRESSION>

<$REFERENCE> ::= <$LEFT-PART> $ <$RIGHT-PART>

<$LEFT-PART> ::= <0> | <ATTRIBUTE> <PV>

<$RIGHT-PART> ::= <0> | [ <VALUE> ] | [ <VALUE> , <VALUE> ]

<FUNCTION-CALL> ::= <IDENTIFIER> < <PARAMETERS> >

<PARAMETERS> ::= <VALUE> | <PARAMETERS> , <VALUE>

<IDENTIFIER> ::= any LISP atom not containing a <DELIMETER>

<S-EXPRESSION> ::= any legal LISP s-expression

<DELIMETER> ::= ! | @ | # | $ | % | ↑ | ( | ) | - | = |
                ~ | < | > | | [ | ] | [] | { | } | {} | ; | : |
                ' | , | . | <OR> | <BLANK>

<OR> ::= the vertical bar "|"

<BLANK> ::= the blank space " "

<0> ::= the empty string, a zeroing out

<RECORD> ::= <RECORD-NAME> ( <CREATION> )
```

```
<RECORD-NAME> ::= <IDENTIFIER>
```

Appendix V

NLP Program for *Paul*

```
COVER ATTR;
vform (main auxil);
main (numb nonfinite pers tense neg);
numb (sing plur);
nonfinite (inf prespart pastpart);
pers (pers1 pers2 pers3);
tense (past present future);
auxil (passive prog perfect);
det (def indef dem possess);
case (common genitive);
common (subjective objective);
roles (agnt aff recip attr dest);
syntaxroles (subject dobject iobject prepobject genitive);
mode (active stative entity);
ending (ed ing);
sub (relative submark)[]

RULES FOR ENCODING;
{agnt is "agent," aff is "affected," recip is "recipient,"
 and attr is "attribute"}

story(para)  -->  paragraph(%top<para(story)>) line story(para:=rest<para>);

story  -->  null;

paragraph(concepts)  -->  concept(%top<concepts(paragraph)>,
                                   ref:=top<concepts(paragraph)>) .
                         paragraph(concepts:=rest<concepts>);
paragraph  -->  null;

concept(↑processed)  -->
        concept(processed,
                focus('previous'):=focus('current'),-focus('current'),
                male('previous'):=
                    list<male('current'),top<male('previous')>>,
                -male('current'),
                female('previous'):=
                      list<female('current'),top<female('previous')>>,
                -female('current'),
                neuter('previous'):=
                      list<neuter('current'),neuter('previous')>>,
                -neuter('current'),
                plur('previous'):=
                    list<plur('current'),top<plur('previous')>>,
                -plur('current'),
                syntaxroles('previous'):=syntaxroles('current'),
                -syntaxroles('current'),
                pronoun('previous'):=pronoun('current'),
                -pronoun('current'));

concept(↑sub)  -->
   concept(submark,idea('previous'):=idea('current'),idea('current'):=ref);
concept(sup=sup(idea('previous')),roles=roles(idea('previous')),
        neg=neg(idea('previous')),↑effect,↑marked)  -->
   concept(marked) , word('too');
 concept(stative,stative(sup))  -->
       clause(%concept,sup:=myrandom<stative(sup(concept))>);
 concept(stative)  -->  clause(%concept,sup:=myrandom<active(sup)>,passive);
 concept(entity)  -->  clause(%concept,'be1',subject:=sup(concept),
                             focus('current'):=sup(concept));
 concept  -->  clause(%concept,sup:=myrandom<active(sup)>,active);

 clause(cause,cause=idea('previous'))  -->  pronoun(sentential) vp(%clause);
 clause(relative)  -->  , pronoun(relative) vp(%clause);
 clause(subject)  -->  np(%subject(clause),ref:=subject(clause),
                          subject('current'):=ref,subject,subjective)
                       vp(%clause,numb:=numb(subject),
                          pers:=pers(subject));
 clause(passive,recip)  -->  clause(subject:=recip,-recip);
 clause(passive)  -->  clause(subject:=aff,-aff);
 clause(stative,cause)  -->  clause(subject:=cause,-cause);
 clause(stative,exp)  -->  clause(subject:=exp,-exp);
 clause(stative)  -->  clause(subject:=recip,-recip);
 clause  -->  clause(subject:=agnt,-agnt);

 np(↑class)  -->  np(class:=classify<ref>);
 np(↑remembered,plur)  -->  np(remembered,plur('current'):=ref);
 np(↑remembered)  -->  np(remembered,@[ref$['gender']]('current'):=ref);
 np(mode)  -->  concept(%np,prespart,sub);
 np(↑member<ref,nouns('said')>,↑flagged)  -->
    np(nouns('said'):=cons<ref,nouns('said')>,flagged);
 np(and,greaterp<length<and>,!2 >)  -->  np(%top<and(np)>,remembered) ,
                                         np(and:=rest<and(np)>);
 np(and)  -->  np(%top<and(np)>,remembered,class:=class(np))
               word('and')
               np(%bottom<and(np)>,remembered,class:=class(np));
 np(class='I')  -->  pronoun(%np,pronoun('current'):=ref);
 np(class='III',pronoun('previous')=ref)  -->
    pronoun(%np,pronoun('current'):=ref);
 np(↑det,↑$['proper'],↑class='NONE')  -->  np(def);
 np(↑det,↑$['proper'])  -->  np(indef);
 np(det,↑detr)  -->  detr(%np)  np(detr);
 np(class='II'|class='III',$['intelligent'],↑flagged)  -->
    np(sup:=randomchain<sup(sup),'intelligent'>,def,flagged,subst);
 np(class='II'|class='III',↑flagged)  -->
    np(sup:=randomchain<sup(sup),!nil>,def,flagged,subst);
 np(subst,ambiguous&superordinate&test<sup,ref>)  -->
    adj(sup:=disambiguate<ref(np),sup(np)>) np(-subst);
 np  -->  noun(%np,-flagged,-remembered);

 vp(aff,↑focus('current'))  -->  vp(focus('current'):=aff);
 vp(recip,↑focus('current'))  -->  vp(focus('current'):=recip);
 vp(agnt,↑focus('current'))  -->  vp(focus('current'):=agnt);
 vp(exp,↑focus('current'))  -->  vp(focus('current'):=exp);
 vp(subject,↑focus('current'))  -->  vp(focus('current'):=subject);
 vp(↑numb,↑nonfinite)  -->  vp(sing);
 vp(↑pers,↑nonfinite)  -->  vp(pers3);
 vp(↑tense,↑nonfinite)  -->  vp(present);
 vp(neg,↑auxil,↑'do1')  -->  vp('do1',-auxil,-roles) vp(-main,inf);
```

```
vp(perfect) --> vp('have1',-auxil,-roles) vp(-main,-perfect,pastpart);
vp(prog) --> vp('be1',-auxil,-roles) vp(-main,-prog,-prespart);
vp(passive) --> vp('be1',-auxil,-roles) vp(-main,-passive,pastpart);
vp(char) --> vp(-char) adj(sup:=char(vp));
vp(effect) --> vp(-effect) concept(%effect(vp),relative);
vp(agnt,agntprep(sup)) --> vp(-agnt)
                           pp(%agnt(vp),ref:=agnt(vp),
                              prep:=agntprep(sup(vp)));
vp(agnt) --> vp(-agnt) pp(%agnt(vp),ref:=agnt(vp),prep:='by');
vp(recip,recipprep(sup)) --> vp(-recip)
                             pp(%recip(vp),ref:=recip(vp),
                                prep:=recipprep(sup(vp)),dobject);
vp(recip,active) --> vp(-recip)
                     pp(%recip(vp),ref:=recip(vp),
                        prep:='to',iobject);
vp(recip) --> vp(-recip)
              np(%recip(vp),ref:=recip(vp),
                 objective,dobject('current'):=ref);
vp(dest,dest=dest(idea('previous'))) --> vp(-dest);
vp(dest) --> vp(-dest)
             pp(%dest(vp),ref:=dest(vp),prep:='to',dest);
vp(aff) --> vp(-aff)
            np(%aff(vp),ref:=aff(vp),objective,
               dobject('current'):=ref);
vp(phrasalprep(sup),↑marked) --> vp(marked)
                                 prep(sup:=phrasalprep(sup(vp)));
vp(neg,$='auxiliary') --> vp(-neg) word('not');
vp --> verb(%vp);

pp(dobject) --> prep(sup:=prep(pp));
                np(%pp,-prep,objective,dobject('current'):=ref);
pp(iobject) --> prep(sup:=prep(pp))
                np(%pp,-prep,objective,iobject('current'):=ref);
pp --> prep(sup:=prep(pp)) np(%pp,-prep,objective);

noun(word(sup)) --> nounp(%noun,sup:=word(sup(noun)));
noun --> nounp(%noun);
verb --> verbp(%verb,sup:=word(sup(verb)));
adj(det) --> detr(%adj) adj(-det);
adj --> word(%adj);
detr(possess,↑class) --> detr(class:=classify<sup>);
detr(possess,class='I') -->
    pronoun(sup:=possess(detr),genitive,
            genitive('current'):=sup,
            @[sup$['gender']]('current'):=sup,
            pronoun('current'):=ref);
detr(possess) --> np(sup:=sup(possess(detr)),ref:=possess(detr),
                     genitive,genitive('current'):=ref)
                  morpheme('possess');
detr(indef,plur) --> null;
detr(indef,vowel(sup)|(↑consonant(sup),vowel<sup>)) --> word('an');
detr(indef) --> word('a');
detr(def) --> word('the');
prep --> word(%prep);
```

```
pronoun(relative)  -->  word('which');
pronoun(sentential)  -->  word('this');
pronoun(pers2,common)  -->  word('you');
pronoun(pers2)  ->  word('your');

pronoun(pers1,plur,subjective)  -->  word('we');
pronoun(pers1,plur,objective)  -->  word('us');
pronoun(pers1,plur)  ->  word('our');
pronoun(pers1,subjective)  -->  word('i');
pronoun(pers1,objective)  -->  word('me');
pronoun(pers1)  -->  word('my');

pronoun(plur,subjective)  -->  word('they');
pronoun(plur,objective)  -->  word('them');
pronoun(plur)  -->  word('their');
pronoun(subjective)  -->  word(sup:=@[$['gender']]('subjective'));
pronoun(objective)  -->  word(sup:=@[$['gender']]('objective'));
pronoun  -->  word(sup:=@[$['gender']]('genitive'));

nounp(plur,plur(sup))  -->  word(%nounp,sup:=plur(sup));
nounp(plur)  -->  word(%nounp) s;
nounp  -->  word(%nounp);

verbp(inf)  -->  word(%verbp);
verbp(past,plur,pastplur(sup))  --> word(%verbp,sup:=pastplur(sup));
verbp(past,past(sup))  -->  word(%verbp,sup:=past(sup));
verbp(pastpart,pastpart(sup))  -->  word(%verbp,sup:=pastpart(sup));
verbp(past|pastpart)  -->  word(%verbp,ed) ed;
verbp(prespart)  -->  word(%verbp,ing) ing;
verbp(plur|pers2,plur(sup))  -->  word(%verbp,sup:=plur(sup));
verbp(plur|pers2)  -->  word(%verbp);
verbp(pers1,pers1(sup))  -->  word(%verbp,sup:=pers1(sup));
verbp(pers1)  -->  word(%verbp);
verbp(pers3,pers3(sup))  -->  word(%verbp,sup:=pers3(sup));
verbp(pers3,es(sup))  -->  word(%verbp) es;
verbp  -->  word(%verbp) s;

word('null')  -->  null;
word(ending,fincon(sup))  -->  # word(-ending)
                                output(sup:=double<sup(word)>);
{FINCON and DOUBLE are for doubling the final consonant}
word(e(sup),↑ending)  -->  # output(%word) e;
word  -->  # output(%word);

morpheme('possess')  -->  ' s[]


RECORDS;

{verbs}
anger\ ('feelbad',stative:=list<'anger1'>);
anger1 ('anger\',word:='anger');
be1 ('auxiliary',word:='be');
break\ ('destroy',active:=list<'break1'>);
break1 ('break\',word:='break);
```

```
buy\ ('acquire',active:=list<'purchase1','buy1'>);
buy1 ('buy\',word:='buy');
care1 ('like',word:='car',recipprep:='for');
cry1 ('cry\',word:='cry');
cry\ ('selfexpress',active:=list<'cry1','weep1'>);
do1 ('auxiliary',word:='do');
drop1 ('drop\',word:='drop',phrasalprep:='off');
drop\ ('tumble',stative:=list<'drop1','fall1'>);
enjoy1 ('enjoy\',word:='enjoy');
enjoy\ ('feel+',active:=list<'enjoy1'>,stative:=list<'please1'>);
fall1 ('drop\',word:='fall',phrasalprep:='off');
give ('transfer',active:=list<'give1'>,stative:=list<'receive1'>);
give1 ('give',word:='giv');
go\ ('move',active:=list<'go1'>);
go1 ('go\',word:='go');
have1 ('auxiliary',word:='hav');
hit1 ('hit\',word:='hit');
hit\ ('phys\abuse',active:=list<'hit1','punch1','slug1'>);
kiss\ ('phys\love',active:=list<'kiss1'>);
kiss1 ('kiss\',word:='kiss');
like ('feel+',stative:=list<'like1','care1'>);
like1 ('like',word:='lik');
please1 ('enjoy\',word:='pleas');
punch1 ('hit\',word:='punch');
purchase1 ('buy\',word:='purchas');
receive1 ('give',word:='receiv',agntprep:='from');
slug1 ('hit\',word:='slug');
take ('acquire',active:=list<'take1'>);
take1 ('take',word:='tak');
upset1 ('upset\',word:='upset');
upset\ ('feelbad',stative:=list<'upset1'>);
want\ ('desire\',stative:=list<'want1'>,active:=list<'lust1'>);
want1 ('want\',word:='want');
weep1 ('cry\',word:='weep');

{nouns}
animal ('living',intelligent,animate);
bird ('animal',skin:='feathered',blood:='warm');
boombah ('chicken',size:='large',color:='red',gender:='female',proper);
chicken ('bird',color:='brown',size:='small');
churchy ('turtle',gender:='male',proper);
hepzibah ('skunk',gender:='female',proper);
howland ('owl',gender:='male',proper);
kite ('toy');
living ('thing');
mammal ('animal',skin:='furry',blood:='warm');
owl ('bird',color:='brown',size:='small');
petal ('living',partof:='rose');
place ('thing');
plant ('living');
pogo ('possum',gender:='male',proper);
possum ('mammal',color:='grey',size:='small');
reptile ('animal',skin:='scaled',blood:='cold');
rose ('plant');
skunk ('mammal',color:='black',feature:='whitestriped',size:='small');
```

```
store ('place');
thing (gender:='neuter');
toy ('thing');
turtle ('reptile',color:='green',feature:='[illegible]',size:='small');

{morphological entries}
be ('be',past:='was',[illegible]:='were',[illegible]:='is',
    plur:='are',[illegible]);
break (past:=[illegible]'broken');
buy (past:='bought',[illegible]'bought');
die (e);
cry (past:='cried',[illegible]'cried',pers3:=[illegible]);
do (past:='did',[illegible]'done',pers3:='does');
drop (Tdouble);
fall (past:='fell',pastpart:='fallen');
give ([illegible]'given');
go (past:=[illegible]'gone');
have ([illegible],pers3:='has');
hit (past:='hit'[illegible]'hit');
kiss (es);
[illegible] (e);
place (e);
punch (es);
[illegible] (e);
[illegible] (es);
[illegible] (Tdouble);
run (e,past:='ran',[illegible]'run');
[illegible]'[illegible]',Tdouble[illegible]
[illegible]

[illegible]
[illegible]
[illegible]
[illegible]

CLOSE IN[]
```

[illegible]

20. Jensen, Karen, and George E. Heidorn. The Fitted Parse: 100% Parsing Capability in a Syntactic Grammar of English. Tech. Rep. RC 9729 (#42958), IBM Thomas J. Watson Research Center, 1982.

21. Kaplan, Ronald M., and Joan W. Bresnan. Lexical Functional Grammar: A Formal System for Grammatical Representation. In *The Mental Representation of Grammatical Relations*, J. W. Bresnan, Ed., The MIT Press, Cambridge, to be published.

22. Katz, Boris. A Three-Step Procedure for Language Generation. Tech. Rep. Artificial Intelligence Memo No. 599, MIT, Cambridge, 1980.

23. Langacker, Ronald W. *Fundamentals of Linguistic Analysis*. Harcourt Brace Jovanovich, Inc., New York, 1972.

24. Leggett, Glenn, C. David Mead, and William Charvat. *Prentice-Hall Handbook for Writers*. Prentice-Hall, Inc., Englewood Cliffs, N.J., 1965.

25. Lehnert, Wendy G. *The Process of Question Answering*. Erlbaum Associates, Hillsdale, N.J., 1978.

26. Levin, James A., and Neil M. Goldman. Process Models of Reference in Context. Tech. Rep. ISI/RR-78-72, Information Sciences Institute, Marina del Rey, Cal., 1978.

27. Luria, Marc. Dividing Up the Question Answering Process. Proceedings of the National Conference on Artificial Intelligence, National Conference on Artificial Intelligence, 1982.

28. Mann, William C. Two Discourse Generators. Proceedings of the 19th Annual Meeting of the Association for Computational Linguistics, Association for Computational Linguistics, 1981.

29. Mann, William C., and James A. Moore. Computer Generation of Multiparagraph English Text. *American Journal of Computational Linguistics 7*, 1 (January-March 1981).

30. Mann, William C., Madeline Bates, Barbara J. Grosz, David D. McDonald, Kathleen R. McKeown, and William R. Swartout. Text Generation: The State of the Art and the Literature. Tech. Rep. ISI/RR-81-101, Information Sciences Institute, Marina del Rey, Cal., 1981. Also University of Pennsylvania MS-CIS-81-9.

31. Matthiessen, Christian M. I. M. A Grammar and a Lexicon for a Text-Production System. Proceedings of the 19th Annual Meeting of the Association for Computational Linguistics, Association for Computational Linguistics, 1981.

32. McDonald, David Daniel. Natural Language Production as a Process of Decision Making Under Constraints. Ph.D. Th., Massachusetts Institute of Technology, 1980.

33. McDonald, David D. Language Production: The Source of the Dictionary. Proceedings of the 19th Annual Meeting of the Association for Computational Linguistics, Association for Computational Linguistics, 1981.

34. McDonald, David D. Natural Language Generation as a Computational Problem: An Introduction. Tech. Rep. COINS Technical Report 81-33, University of Massachusetts at Amherst, 1981.

35. McKeown, Kathleen Rose. Generating Natural Language Text in Response to Questions about Database Structure. Ph.D. Th., University of Pennsylvania, 1982.

36. Minsky, Marvin. A Framework for Representing Knowledge. Tech. Rep. Artificial Intelligence Memo No. 306, MIT, Cambridge, 1974.